In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

University of Alberta

Shear–Warp Factorization Volume Rendering Visualization System

by

Nataliya V. Kazakova ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment
of the requirements for the degree of **Master of Science**

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87798-1
Our file Notre référence
ISBN: 0-612-87798-1

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Nataliya V. Kazakova

Title of Thesis: **Shear - Warp Factorization Volume Rendering Visualization System**

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for privat, scholarly or scientific research pupose only.

The author reserve all other publications and other rights in assosiation with the copyright in the thesis, and except as herein before provided, neither the thesis nor any subsrantial portion thereof may be printed or otherwise reproduced in any material from whatever without the author's written permission.

P

July 4th 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommended to the **Faculty of Graduate Studies and Research** for acceptance, a thesis entitled **Shear - Warp Factorization Volume Rendering Visualization System** submitted by Nataliya V. Kazakova in partial fulfillment of the requirements for the degree of Master of Science

Dr. Nelson G. Durdle, co-supervisor

Dr. Duncan Elliott

Dr. James Hoover

July 4th 2003

Abstract

Volume rendering is a computationally intensive DSP application where three-dimensional volumetric data is directly converted and displayed as a two-dimensional image. In order for scientists and engineers to effectively use volume rendering and visualization as a tool, interactive full-frame visualization is essential. Therefore, the target of the volume rendering is building an interactive real-time system while providing an accurate image. The current thesis describes a Volume Rendering System using the Shear-Warp Factorization algorithm (SWF).

Shear-warp factorization algorithm has been chosen because in comparison to other direct volume rendering algorithms it is the fastest. The SWF algorithm implemented in hardware allows real-time rendering due to the time slicing approach, which provides fast access to the components, and the system re-configurability.

The problem of rendering speed was solved on four hierarchical levels: the system level, component level, circuit level and logic level. The power consumption optimization was applied to the circuit and logic level. Fast and low power logic based on Non-Full Swing Complementary Pass Transistors Logic was designed and tested.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisors, Dr. Nelson G. Durdle and Dr. Martin Margala, for their patience, guidance, assistance, constant support and sincere friendship that provided an environment conducive to research and learning.

My special appreciation and thanks to Dr. Igor Filanovsky for his invaluable advises and help during my time at the University of Alberta.

I want to acknowledge the support of Canadian Microelectronics Corporation (CMC) and Micronet without which this research could not have been accomplished.

Special thanks go to the Department of Electrical and Computer Engineering for its role in the financial support of my research.

I am also very grateful to Raymond Sung for his help and Julien Lamoureux for his contribution to work over the summer.

In addition, I would like to thank the PMC-Sierra, Inc. and, especially, Tim Parsons, the project leader, for support and opportunity to learn what a design engineer should be.

Last but not least appreciation goes to my family for their steadfast faith in my abilities and myself.

Table of Content

CHAPTER 1 1

1. VOLUME RENDERING 1

1.1. INTRODUCTION 1

1.1.1. Volume Rendering Advantages 2

1.1.2. Volume Rendering Pipeline 4

1.1.2.1. Segmentation 4

1.1.2.2. Gradient Computation 5

1.1.2.3. Resampling 5

1.1.2.4. Classification 5

1.1.2.5. Shading 5

1.1.2.6. Compositing 6

1.1.3. Volume Rendering Algorithms 6

1.1.3.1. Algorithms Descriptions 6

1.1.3.2. Algorithms Analysis 7

1.1.3.3. Shear – Warp Factorization Algorithm 9

1.1.3.4. Mathematical Aspects of Shear-Warp Factorization Algorithm 12

1.1.3.5. Shear-Warp Factorization Rendered Examples 18

1.1.4. Thesis Organization 21

CHAPTER 2 23

2. VOLUME RENDERING - STATE OF THE ART 23

2.1. VOLUME RENDERING SYSTEMS 23

2.1.1. Software Implementations 23

2.1.2. Hardware Implementations 24

2.1.2.1. CUBE Project 25

2.1.2.2. VIZARD II 26

2.1.2.3. VolumePro 27

2.1.3. Chapter Summary 28

CHAPTER 3 30

3. SHEAR – WARP FACTORIZATION VOLUME RENDERING SYSTEM . 30

3.1. SYSTEM DESIGN 30
  3.1.1. Design Flow 30
  3.1.2. VRS Time-Slicing Architecture 33
    3.1.2.1. External Communication Interface 34
    3.1.2.2. Context RAM 36
    3.1.2.3. Testability 39
    3.1.2.4. Pipeline Stages 41
      3.1.2.4.1. Pipelining and clock tree 42
  3.1.3. Chapter Summary 43

CHAPTER 4 44

4. SWF VRS COMPONENTS DESIGN 44

4.1. GRADIENT PROCESSOR 44
  4.1.1. Gradient Operators 44
  4.1.2. Sobel Operator 46
    4.1.2.1. Mathematical Derivations 48
    4.1.2.2. Previous Work 53
  4.1.3. Sobel Operator Implementation 53
    4.1.3.1. Sobel Edge Detection Processor Architecture 55
      4.1.3.1.1. Design of the Memory Processor 57
      4.1.3.1.2. Design of the Window Processor 58
      4.1.3.1.3. Design of the Magnitude Processor 60
  4.1.4. Sobel Operator Processor Verification 63
4.2. MATRIX MULTIPLICATION PROCESSOR 66
  4.2.1. Application Aspects 66
  4.2.2. MMP Design and Verification 68
4.3. SHEARER – COMPOSER 75
  4.3.1. Shearer- Composer Mathematical Model 75

4.3.2. Shearer – Composer Design and Verification 77
4.3.2.1. Data Path 79
4.3.2.2. Control Path 79
4.3.2.3. Testing strategy for Shearer - Composer 79
4.3.2.4. Implementation in MAX 7128SLC84-10 FPGA 81
4.4. SHADING PROCESSOR 84
4.4.1. Shading Technique *84*
4.4.1.1. Phong Shading 86
4.4.1.1.1. The Phong illumination model 86
4.4.1.1.2. The Phong shading model 86
4.4.1.2. Mathematical Derivations 88
4.4.1.2.1. Angle approximation? 90
4.5. CLASSIFIER 92
4.5.1. Chapter Summary *93*

CHAPTER 5 **94**

5. CIRCUIT DESIGN AND VERIFICATION **94**

5.1. INNER PRODUCT PROCESSOR 94
5.1.1. Applicatory Aspects *94*
5.1.2. IPP Architecture Based on the Merged Arithmetic *95*
5.1.3. Modified Booth Encoder *96*
5.1.3.1. Encoder 96
5.1.3.2. Partial Product Generator 96
5.1.3.3. Two's Complement Adder 97
5.1.3.4. Sign Extension Simplification 98
5.1.3.5. 3-D Booth Encoder 99
5.1.4. Wallace Reduction Tree *100*
5.1.4.1. First Stage/Level 1 103
5.1.4.2. Second Stage/Level 2 103
5.1.4.3. Third Stage/Level 3 103
5.1.5. Adder *103*

5.1.6. Final Two Operands Carry-Free Adder 105
5.1.7. Inner Product Processor Design Conclusion 106
5.1.8. Circuit Design Verification 107
5.1.8.1. Booth Multiplier Design Verification 107
5.1.8.1.1. Booth Selector Verification 107
5.1.8.1.2. Partial Product Generator Cell Design Verification 108
5.1.8.1.3. Modified Booth Encoder Design Verification 108
5.1.8.2. 15-bit Carry Free Adder Design Verification 108
5.1.8.3. Inner Product Processor Design Verification 109
5.1.9. Chip Fabrication and Evaluation 111
5.1.9.1. Packaging 112
5.1.9.2. Chip Testing 113
5.1.9.2.1. Test System 113
5.1.9.2.2. Testing Results 113
5.1.9.2.3. Error detection 114
5.1.9.2.4. Errors Correction 115
5.1.10. Circuit VHDL Implementations 115
5.2. LOGIC DESIGN 116
5.2.1. Low Power Logic Design 116
5.2.2. Non-Full Swing Pass Transistor Logic (NFS CPL) Cells 116
5.2.2.1. Process and Device Parameter 117
5.2.3. Basic Gate Design 118
5.2.3.1. 4-2 Compressors 119
5.2.3.1.1. 4-2 Compressor Design 121
5.2.3.1.2. 4-2 Compressor Design Verification 124
5.2.3.1.3. Flip Flop 125
5.2.3.1.4. PTFF Design Verification 127
5.2.4. Chapter Summary 127

CHAPTER 6 129

6. CONCLUSIONS AND FUTURE WORK 129

6.1. FINAL SUMMARY 129
6.2. FUTURE WORK 131
6.2.1. Future Directions for the VRS Performance Improvements 131

REFERENCES 132

APPENDIX 140

GLOSSARY 146

Table of Figures

FIGURE 1 VOLUME RENDERING VS. SURFACE RENDERING 3
FIGURE 2 VOLUME RENDERING PIPELINE 4
FIGURE 3 SHEAR-WARP FACTORIZATION ALGORITHM 9
FIGURE 4 OBJECT SLICING IN THE VIEWING DIRECTION 10
FIGURE 5 OBJECT COORDINATE SYSTEM 10
FIGURE 6 STANDARD OBJECT COORDINATE SYSTEM 11
FIGURE 7 SHEARED OBJECT COORDINATE SYSTEM 11
FIGURE 8 PARALLEL PROJECTIONS 13
FIGURE 9 PERSPECTIVE PROJECTIONS 13
FIGURE 10 INTERMEDIATE IMAGE COORDINATE SYSTEM 15
FIGURE 11 SHEAR - WARP FACTORIZATION ALGORITHM 18
FIGURE 12 HUMAN HEAD IMAGE RENDERED USING SHEAR-WARP FACTORIZATION ALGORITHM 19
FIGURE 13 VOLUME CONSISTENT SLICES IN Z-AXIS DIRECTION 20
FIGURE 14 CLASSIFICATION EXAMPLE OF THE RENDERED IMAGE 20
FIGURE 15 LIGHTENED MODEL 21
FIGURE 16 ARCHITECTURE OF VIZARD II PCI BOARD 26
FIGURE 17 VIZARD II RAY-CASTING ENGINE 26
FIGURE 18 VOLUMEPRO ARCHITECTURE 27
FIGURE 19 DESIGN DEVELOPMENT DATA FLOW 30
FIGURE 20 SWF VRS STRUCTURE 33
FIGURE 21 COMMUNICATION INTERFACE STRUCTURE 35
FIGURE 22 CONTEXT RAM SNOOPING ADDRESS FORMATION 36
FIGURE 23 DUAL PORT CONTEXT RAM PORTS 38
FIGURE 24 MICROPROCESSOR TO CONTEXT RAM ACCESS STATE MACHINE 38
FIGURE 25 INSERTING TESTABILITY BY INSERTING MULTIPLEXERS 39
FIGURE 26 EMBEDDED BIST 39
FIGURE 27 BIST USING TEST BENCHES 40
FIGURE 28 SWF VRS BIST 41
FIGURE 29 PIPELINING STAGES STRUCTURE 41

FIGURE 30 PIPELINING AND MULTI-CLOCKING SOLUTION 42
FIGURE 31 CLOCK FREQUENCY DIVIDER 43
FIGURE 32 APPROXIMATED SIGNAL 44
FIGURE 33 GRADIENT OF THE SIGNAL 45
FIGURE 34 TYPICAL ARRANGEMENT OF A WINDOW OF PIXELS FOR AN IMAGE 47
FIGURE 35 SOBEL CONVOLUTION KERNELS 47
FIGURE 36 WINDOW UPDATING 47
FIGURE 37 DIRECTION REPRESENTATION 48
FIGURE 38 THE RESULTS OF CORRESPONDING SOBEL CONVOLUTION KERNELS APPLICATIONS 49
FIGURE 39 EUCLIDIAN NORMAL APPROXIMATION 50
FIGURE 40 ERROR FOR APPROXIMATING THE MAGNITUDE FUNCTION 52
FIGURE 41 EDGE DETECTION BOUNDARIES 53
FIGURE 42 FUNCTIONAL BLOCK DIAGRAM OF THE MAGNITUDE AND DIRECTION PROCESSOR 54
FIGURE 43 TOP LEVEL DECOMPOSITION OF THE SOBEL EDGE DETECTION SYSTEM 55
FIGURE 44 HIERARCHICAL DECOMPOSITION OF THE SOBEL OPERATOR EDGE DETECTING SYSTEM 56
FIGURE 45 MEMORY ARRAY 57
FIGURE 46 VOLUME SLICE (IMAGE) WINDOW 57
FIGURE 47 BASIC SOBEL OPERATOR CELL 58
FIGURE 48 FILTER MAGNITUDE CALCULATOR BLOCK DIAGRAM 59
FIGURE 49 SOBEL OPERATOR MAGNITUDE AND DIRECTION PROCESSOR BLOCK DIAGRAM 61
FIGURE 50 THE IMAGE 64
FIGURE 51 SOBEL EDGES OF THE IMAGE 64
FIGURE 52 SHEAR-WARP FACTORIZATION COORDINATE TRANSFORMATION 66
FIGURE 53 MATRIX MULTIPLICATION PROCESSOR STRUCTURE 67
FIGURE 54 MMP CONTROLLER DIAGRAM 69
FIGURE 55 16-BIT 3D OPERANDS IPP WITH PARTIALLY MERGED TECHNIQUE 71
FIGURE 56 1-2 MPP FUNCTIONAL VERIFICATION: OUTPUTS 73
FIGURE 57 1-2 MMP FUNCTIONAL SIMULATION: INPUTS 74

FIGURE 58 SWF COMPOSER STRUCTURAL DIAGRAM 76
FIGURE 59 SHEARER-COMPOSER FLOW CHART 77
FIGURE 60 DATA PATH STRUCTURAL SCHEMATIC 78
FIGURE 61 COMPOSER FUNCTIONAL SIMULATION: NON-TRANSPARENT VOXELS' COMPOSITING 80
FIGURE 62 TRANSPARENT VOXEL SKIPPING 81
FIGURE 63 SHADING STAGE IN VOLUME RENDERING PIPELINE 84
FIGURE 64 PHONG SHADING PARAMETERS 87
FIGURE 65 BELA VRS SHADER SCHEMATIC 87
FIGURE 66 OPACITY WEIGHTED COLOR INTERPOLATION PIPELINING 92
FIGURE 67 ARTIFACTS OF SEPARATE INTERPOLATION OF COLOR AND OPACITY 93
FIGURE 68 BOOTH ENCODER BASIC CELL 96
FIGURE 69 PARTIAL PRODUCT GENERATOR CELL 97
FIGURE 70 TWO'S COMPLIMENT ADDER CELL 97
FIGURE 71 TWO'S COMPLIMENT CARRY SELECT ADDER 98
FIGURE 72 SIGN EXTENSION SIMPLIFICATION CELL 99
FIGURE 73 SECTION OF WALLACE TREE USING 4-2 COMPRESSORS 101
FIGURE 74 WALLACE TREE BIT PRODUCT REDUCTION 102
FIGURE 75 REDUNDANT BINARY ADDITION 104
FIGURE 76 CARRY-FREE ADDER CELL 105
FIGURE 77 FINAL CARRY SELECT ADDER BLOCK DIAGRAM 106
FIGURE 78 (A,B,C) INNER PRODUCT PROCESSOR PERFORMANCE RESULTS 111
FIGURE 79 INNER PRODUCT PROCESSOR CORE LAYOUT 111
FIGURE 80 INNER PRODUCT PROCESSOR CHIP LITHOGRAPHY 112
FIGURE 81 TRANSISTORS SIZE DEPENDENCE 118
FIGURE 82 NFS AND FS LOGIC CELLS 119
FIGURE 83 (A,B) 4-2 COMPRESSOR TG BASED 120
FIGURE 84 TG BASED 4-2 COMPRESSOR PERFORMANCE IN 0.18 UM CMOS 121
FIGURE 85 TG BASED 4-2 COMPRESSOR PERFORMANCE IN 0.13 UM CMOS 121
FIGURE 86 NFS 4-2 COMPRESSOR 122
FIGURE 87 FULL-SWING AND NON-FULL-SWING MULTIPLEXERS 123

FIGURE 88 4-2 COMPRESSOR FUNCTIONAL TIMING DIAGRAM.. 124
FIGURE 89 PTFF SCHEMATIC.. 125
FIGURE 90 PULSED CLOCK GENERATOR 126
FIGURE 91 PTFF FUNCTIONAL TIMING DIAGRAM.. 127

Table of Tables

TABLE 1 BENCHMARK DATASETS 7

TABLE 2 AVERAGE FRAME TIMES (IN SEC) 8

TABLE 3 CONTEXT RAM 36

TABLE 4 CODES FOR EDGE DETECTION 48

TABLE 5 THE IMAGE EDGES COMPUTED BY SOBEL EDGE DETECTOR 65

TABLE 6 FILTER CIRCUIT PERFORMANCE 65

TABLE 7 PIPELINING TASKS SCHEDULING 70

TABLE 8 ARITHMETIC OPERATIONS FOR PHONG SHADING WITH A SINGLE LIGHT SOURCE (N IS A SPECULAR EXPONENT) 88

TABLE 9 REDUNDANT BINARY ADDITION ENCODING RULE 104

TABLE 10 ADDING RULE 104

TABLE 11 15-BIT ADDER TEST VECTORS 108

TABLE 12 SAMPLE ERRORS (DOT_PRODUCT SHOULD EQUAL 0) 114

TABLE 13 4-2 COMPRESSORS REVIEW 119

TABLE 14 4-2 COMPRESSOR TEST VECTORS 124

TABLE 15 LIBRARY DEPENDENT FF PARAMETERS 126

Chapter 1

1. Volume Rendering

1.1. Introduction

The rapid development of Computer Tomography (CT) has resulted in new CT applications. One of these applications, three-dimensional (3D) CT with volume rendering, is now a major area of clinical and academic interest. Volume rendering generates accurate and immediately available images from the scanning systems such as CAT (Computed Axial Tomography), MRI (Magnetic Resonance Imaging) or LCM (Laser Confocal Microscopy), or directly from three-dimensional simulations of data without extensive editing. It allows a specialist to explore specific problems within different aspects of the data set.

All 3D rendering techniques represent a 3D volume of data in one or more two-dimensional (2D) planes, conveying the spatial relationships inherent in the data with use of visual depth cues. The data are organized into a 3D matrix of volume elements (voxels). All 3D rendering techniques rely on mathematical formulas to determine for each pixel what portion of the data in memory should be displayed on the screen and how that portion should be weighted to best represent spatial relationships. Voxel selection is usually accomplished by projecting rays through the data set that correspond to the pixel matrix of the desired 2D image. Differences in the images produced with various 3D rendering techniques are the result of variations in how voxels are selected and weighted.

3D volume rendering takes the entire volume of data, sums the contributions of each voxel along a line from the viewer's eye through the data set, and displays the resulting composite for each pixel of the display. Incorporation of information from the entire volume can lead to greater fidelity to the data; however, powerful and expensive computers are required to perform volume rendering at a reasonable speed.

Volume rendering is being used in a wide variety of applications ranging from seismic data display to wind tunnel testing. Now volume rendering is being incorporated mainly into commercially available software packages and very few hardware implementations for medical imaging.

This thesis presents a new real- time volume rendering system with low computational costs and high performance.

The rest of this chapter briefly describes the volume rendering advantages versus to polygonal rendering; explains the rendering pipeline, steps and components common for the major volume rendering algorithms, provides the rendering parameters specific each of the rendering stages and gives algorithms descriptions and comparisons. It also explains the reason why the shear-warp factorization algorithm has been chosen for the implementation, and, finally, shows the thesis organization.

1.1.1. Volume Rendering Advantages

Advantages of surface rendering include superior speed and flexibility in image rendering [32]. There are some important advantages that volume graphics have over surface-based techniques, and also some disadvantages [66].

Advantages:

- The most important advantage is that volume graphics provide object structure and tissues diversity.
- Another advantage is the fact that any viewpoint-independent characteristic of the voxel can be stored along with it in the volume buffer. This cannot be done with surface graphics since surfaces are polygonized and are not represented by the actual voxels themselves.
- The most important advantage relevant to medical applications is that the interior of the object is accessible to the viewer. This advantage is invaluable in surgical planning allowing surface models to be interactively repositioned and manipulated. An example is given below.

The images of a femur, shown in Figure 1 were generated from the same 3D MRI knee image. Image (c) was created using volume rendering with shading. Image (a) was created by polygon rendering a surface model that was built using the dividing cubes algorithm followed by polygon reduction and smoothing. Both images provide good quality visualization of the bone surface.

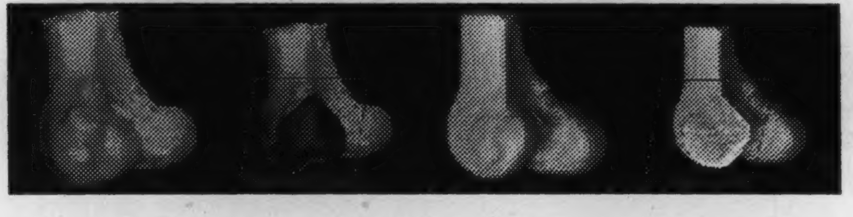

(a) (b) (c) (d)

Figure 1 Volume rendering vs. surface rendering

Between surface and volume models can be seen in these two images, where the models were cut by the red cutting plane. The cut volume model, image (d), reveals interior structure of the bone but the surface model, image (b) is only a hollow shell. The object's interior structure is especially important for modeling the deformation of soft tissues or cutting through solid objects.

Disadvantages:

- The loss of the continuous form of surface graphics that means that there is a loss of accuracy in the information.
- The biggest obstacle is the cost and technology involved in volume rendering.

Typical volume data sets have volume buffers of the size 512x512x512. This is over 100 million voxels. With just 1 byte per voxel this requires over 128Mbytes. The processing power required to handle the volume buffer is significant. At present the fastest rendering time for a volume buffer of size 256x256x256 (on a single processor system) is approximately one second (conducted on an R4000 Indigo). If acceptable frame rates (10-15 frames per second minimum; 30 frames per second ideally) for real-time animation purposes (copious applications in medicine, e.g. real-time acquisition and viewing of ultrasound data) are to be achieved, there needs to be a thirty-fold improvement in speed over current technology. Even with consideration of this last point, it is apparent that there are enormous advantages to be gained, in terms of medical imaging, through the employment of volume rendering.

1.1.2. Volume Rendering Pipeline

Common steps for the all volume rendering algorithms are summarized in the pipeline in Figure 2 when each operation is completed the data is passed to the next operation.

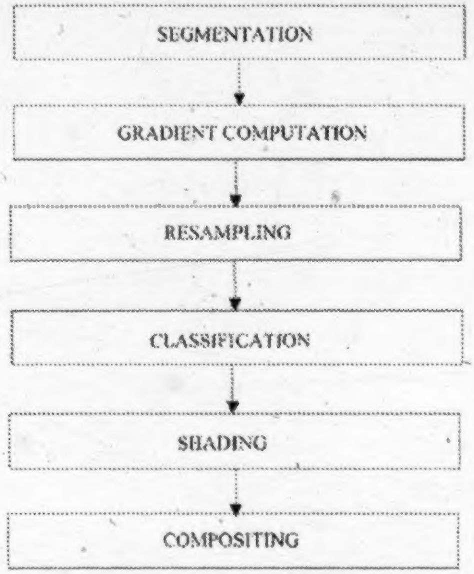


Figure 2 Volume rendering pipeline

The operations in the pipeline consist of segmentation, gradient computation, resampling, shading and compositing. The order and inclusion of those steps vary among volume rendering implementations.

1.1.2.1. Segmentation

Segmentation is not an objective of the current work; we consider this to be done before the data enter the system. Segmentation labels voxels in a data set that is used in the classification stage.

Segmentation is a pre-processing step, which needs to be done just once before actual rendering. This task—a complex analysis of size, shape, intensity, location, texture, and proximity to surrounding structures—is extremely difficult to automate. Besides, there are no general computer segmentation algorithms. Hence, segmentation of images is often only semi-automated, requiring additional manual editing by an expert.

1.1.2.2. Gradient Computation

The gradient computation finds edges or boundaries between different materials. The gradient is a measure of how quickly the data in the data set changes. This information is used in the classification and shading stages. The gradient processor, designed for the Shear-Warp Volume Rendering System is described in the Chapter 4.

1.1.2.3. Resampling

Resampling or interpolation is accomplished with an algorithm that maps a volume data set into a new, often smaller volume to reduce the image processing time. User settings for image quality determine the amount of resampling the computer will perform on the volume data set prior to rendering. Axial sections are scaled in the x and y -axes to reduce the size of the data set (e.g, from 512 x 512 to 256 x 256). Resampling in the z- axis may be arbitrarily set to any suitable value.

1.1.2.4. Classification

Classification is assigning different opacities and colors to voxels, it facilitates seeing inside an object and exploring its structures instead of only visualizing the surface of an object. This is a powerful part of the volume rendering pipeline. This component is described in the Chapter 4.

1.1.2.5. Shading

Shading is used to highlight parts of the data set by employing an illumination model. The major lighting models are described in the Chapter 4.

Simple lighting models construct shading algorithms from computations of the orientation of surfaces relative to a single, fixed light source. More complex models account for multiple light sources, reflections, and shadows within the rendered view. In volume rendering these complex models are seldom used because they result in marginal improvement in comprehension relative to the significantly increased costs for the computing power required. The Shader is described in the Chapter 4.

1.1.2.6. Compositing

Compositing is the way of accumulating sampled values into one volume. There are two basic approaches: front-to-back and back-to-front, it depends on which direction the ray is traversed. In this thesis the composer is the main system component, and is described in detail in the Chapter 4.

1.1.3. Volume Rendering Algorithms

There are several algorithms developed for volume rendering. The volume data is given as a scalar array of sampled points $f(x)$, where $\boldsymbol{x} = (x_i, y_i, z_i)$, that is assumed to be sampled above the Nyquist rate (or filtered). There are basically two ways a volume rendering algorithm can look at the data. Some algorithms view the sampled points as the centers of regions of influence. The contribution of the sample point $\boldsymbol{x}$ to the volume is considered to decrease with distance. One can associate a filter function with each sample point $\boldsymbol{x}$ to represent the contribution of $\boldsymbol{x}$ in the region of influence. This is done in the splatting algorithm. Other algorithms, such as ray casting and shear-warp factorization consider the sample points as the corners of a cell. These reconstruction methods try to guess the values between the sample points by an interpolation method.

1.1.3.1. Algorithms Descriptions

In ray tracing methods a ray starting in the center of an image pixel is directed perpendicular to the image plane away from the viewer. If the ray intersects the volume, the volume is sampled at regular intervals along the ray. The reconstruction of the volume data at a ray sample point is typically performed by tri-linear interpolation of the eight voxels surrounding the ray sample.

Splatting algorithms are a class of rendering algorithms that compute the contribution of a voxel to the image by convoluting the voxel with a filter that distributes the voxel's value to a neighborhood of pixels. This process is analogous to taking a cube of gelatinous material and throwing it onto the image plane. The 'splat' that the gelatin leaves on the image plane is its *footprint*, and determines the voxel's contribution to the final image.

This class of algorithms suffers from the time consuming computations necessary to compute the view dependent filters with which to convolute the voxels in the volume.

Volume shearing methods treat the volume as a stack of 2-D slice images. Each 2-D image is warped from volume space to image space and projected onto the output image. The warping involves a re-sampling of the slice with a linear kernel. It can be performed in two 1-D warping steps: first warping the slice so that its Y coordinate is aligned with the output image's Y coordinates and then warping so that the X coordinates match. Once the slices are warped into the proper coordinate system, they can be blended into the output image.

The ray casting, splatting and volume shearing algorithms of volume renderers differ from each other in the way each one determines the voxel-to-pixel mapping. There are two parts of that process, transforming the voxels from the volume coordinate system to the image coordinate system and reconstructing the voxels into pixels.

1.1.3.2. Algorithms Analysis

The paper by Meissner et al. [59] gives the practical evaluation and comparison of the aforementioned algorithms. The benchmark datasets shown in Table 1 were used for the evaluation in this paper [59].

Table 1 Benchmark Datasets [1]

Dataset	Size	Relevant voxels	Compactness	Pixel content
Blood vessel	256x256x256	79,442 (0.5%)	low	low
Neghip	64x64x64	207,872 (79.3%)	high	medium
Skull	256x256x256	1,384,817 (8.2%)	medium	low
Fuel injection	64x64x64	32,768 (12.5%)	high	medium
Shockwave	64x64x512	1,245,184 (59%)	high	high

---

[1] These datasets, transfer functions, and viewing parameters are publicly available on the internet at http://www.volvis.org

All algorithm-independent image synthesis parameters, such as the viewing matrix, transfer functions, and optical model, were kept constant to enable a fair comparison of the rendering results. Both image quality and computational complexity were evaluated and compared. All presented results were generated on an SGI Octane (R10000 CPU, 250MHz) with 250 MB main memory.

Table 2 shows the average rendering frame rates for 24 random views onto the five datasets for ray casting, splatting and shear-warp algorithms. From the Table 2 we observe that the frame times for the shear-warp are always substantially faster than those of ray casting and splatting in most cases by an order of one or two magnitudes. The shear-warp consistently achieves frame times in the sub-second range.

Table 2 Average frame times (in sec)

	Fuel	Neghip	Skull	Blood	Shock
Ray casting	4.96	8.15	7.78	12.31	3.02
Splatting	1.41	7.35	11.09	1.87	21.77
Shear-warp	0.09	0.24	0.27	0.09	0.91

To evaluate the rendered image quality the visual quality assessment was employed. Meissner et al. in [59] simply put images of competing algorithms side by side, appointing the human visual system (HVS) to be the judge. Since after all images are produced for the human observer and not for error functions, therefore the visual comparison in this case is more appropriate than the numerical.

The observations showed that overall splatting offers a rendering quality similar to that of ray casting, and both ray casting and splatting provided in some cases better image qualities than shear-warp. It can be explained that (as shown in [62]) in the shear-warp algorithm, as it was proposed by Levoy and Lacroute in [3], rendered voxels were interpolated without prior opacity and color being incorporated. This fact has been taken into consideration, and is modified in Chapter 4 as color and opacity weighted interpolation.

The rendering algorithms evaluation and comparison results showed that despite continued advances in volume rendering technology, the Shear-Warp algorithm, although conceived as early as 1994, still remains the world's fastest purely software-based volume rendering algorithm. The impressive speed of near double-digit frame rates for moderately sized datasets, however, does come at the price of reduced image quality and memory consumption.

Very recent research [61] has shown that only rendering via custom hardware could achieve interactive, or real-time, frame rates in excess of 30 frames/s while still providing excellent image quality. Geared to the purpose of providing real-time volume rendering speed the Shear- Warp Factorization algorithm has been chosen for the hardware implementation. A detailed description of this algorithm is given below.

1.1.3.3. Shear – Warp Factorization Algorithm

A functional diagram of the Shear-Warp Factorization algorithm is shown in Figure 3. Proposed by Levoy and primary implemented by Lacroute in [3] the Shear-Warp Factorization algorithm can realize semi-interactive rendering (<10 frames per second (fps)) of volume data with moderate size without affecting image quality on a conventional graphics workstation. However, even utilizing an optimized shear-warp factorization algorithm, volume rendering of a large volume data set is still time consuming and it is very difficult to realize the interactive rendering (10 –15 fps) rate on a single-processor workstation.

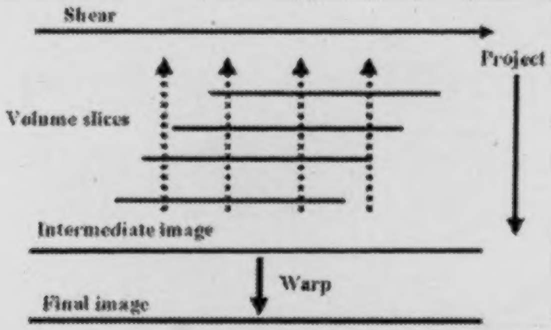


Figure 3 Shear-warp factorization algorithm

The Shear-Warp factorization algorithm factors the viewing transform into a shearing matrix and a warping matrix. The advantage of this operation is that the 3- dimensional (3D) operation of ray casting is transformed into two 2- dimensional (2D) operations.

The resampling, done during the *projection* stage, uses bi-linear interpolation instead of tri-linear interpolation, and the *warping* that is applied to the intermediate image to undo the distortion caused by the shear is another inexpensive 2D operation. As a result of the shearing, the data slices can be accessed in a scanline order, and in synchrony with the scanlines in the intermediate image. Lacroute's implementation of the Shear-Warp algorithm in Figure 3 utilizes a run-length encoding data structure to take advantage of this scanline traversal to further optimize the rendering speed. The resulting algorithm can render 256^3 volumes in under 1 second on an SGI machine without graphics accelerators. Shear - warping is done in four steps:

- *slicing* the original 3-D image into set of 2-D sub-images (Figure 4, Figure 5),
- *shearing* the original object space into shear-space (Figure 6),
- *projecting* the shear-space into an intermediate image (Figure 7),
- *warping* the intermediate image to the final image.

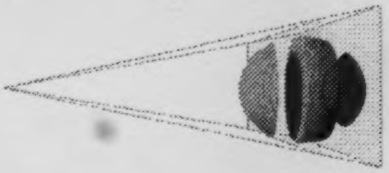


Figure 4 Object slicing in the viewing direction

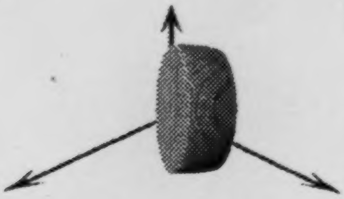


Figure 5 Object coordinate system

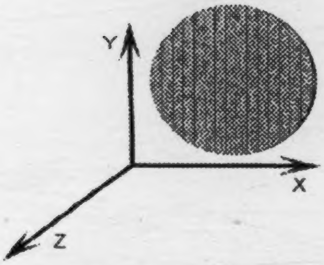


Figure 6 Standard object coordinate system

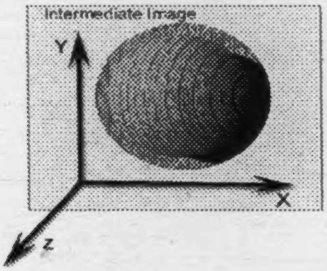


Figure 7 Sheared object coordinate system

The parallel slices of the volume dataset can be viewed upon with a certain viewpoint. During the first step of the shear-warp factorization each slice is sheared individually, dependent on the given viewpoint. For a perspective transformation, the slices have to be scaled to guarantee the perspective effects. This results in the shear-space, which has the interesting property that, by definition, all viewing rays are *parallel* to the third coordinate axis. Another property is that the 3D dataset has now been calculated for the viewpoint, and, therefore, doesn't have to be calculated again for changes in parameters like lighting.

After the shearing, the shear-space is projected into an intermediate, distorted image. The projection is *simple* and *efficient* because of the described above property of the shear-space. The resulting image can simply be mapped (or warped) into the final image. Because this mapping is done on an image (2D), shear - warping has an advantage on other methods that map object-space (3D) into image-space (2D). Of course, viewing transformations/mappings from 2D to 2D require *less computational time* and *memory*, than from 3D to 2D.

1.1.3.4. Mathematical Aspects of Shear-Warp Factorization Algorithm

As shown in Chapter 4 in the Figure 52 sheared and finally warped object involves four coordinate systems. The ojbect coordinate is the original volume coordinate system, where it is sliced, and slice by slice "passed' from system to system untill, finally, is warped into image coordinate space.

To form the standard object coordinate system the viewing axis becomes the object space axis; that is achieved by permutting the object coordinate axis. The sheared object coordinate is formed by shearing the standard coordinate system with sheared matrix; it is also called the intermediate image coordinate system. Finally, a warping matrix transforms the intermediate image into the image coordinate data. Mathematically this procedure might be presented as the following [75] [5]: an Affine viewing transformation matrix *Mview* includes a permutation *P* (4 x 4) matrix, which is the transformation matrix from the object coordinate system into the standard object coordinate system, a 3D shearing (4x4) matrix *Mshear* and a 2D (3 x 3) warping matrix *Mwarp*. The goal of the derivation is to factor *Mview* as follows:

$$Mview = Mwarp * Mshear \qquad \textbf{Equation 1}$$

where

$$Mview = \begin{bmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ M41 & M42 & M43 & M44 \end{bmatrix} \qquad \textbf{Equation 2}$$

and

$$\begin{bmatrix} Xi \\ Yi \\ Zi \\ Wi \end{bmatrix} = Mview * \begin{bmatrix} X0 \\ Y0 \\ Z0 \\ W0 \end{bmatrix} \qquad \textbf{Equation 3}$$

The parallel projections, shown in Figure 8, comply with the next equation:

$$Mview = Mwarp * Mshear * P \qquad \textbf{Equation 4}$$

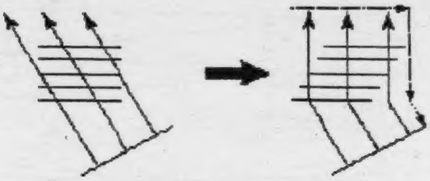


Figure 8 Parallel projections

For the perspective projections, shown in Figure 9 the equation above will be

$$Mview = Mwarp * Mshear * Tshift * P \qquad \textbf{Equation 5}$$

where- Tshift defined by the eye location in the object space.

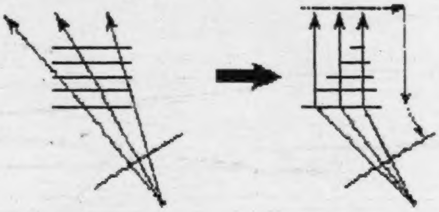


Figure 9 Perspective projections

Both parallel and perspective projections are required. Therefore, Equations 4 and 5 are necessary for the shear-warp factorization transformations.

Thus
$$\begin{bmatrix} i \\ j \\ k \\ 0 \end{bmatrix} = P * \begin{bmatrix} x0 \\ y0 \\ z0 \\ 0 \end{bmatrix} \qquad \textbf{Equation 6}$$

Where
$$P = \begin{bmatrix} p11 & p12 & p13 & 0 \\ p21 & p22 & p23 & 0 \\ p31 & p32 & p33 & 0 \\ p41 & p42 & p43 & 1 \end{bmatrix} \qquad \textbf{Equation 7}$$

Therefore
$$\begin{aligned} i &= p11 * x0 + p12 * y0 + p13 * z0; \\ j &= p21 * x0 + p22 * y0 + p23 * z0; \\ k &= p31 * x0 + p32 * y0 + p33 * z0. \end{aligned} \qquad \textbf{Equation 8}$$

A permutation matrix is defined by the viewing matrix. The principal viewing axis is defined by the smallest angle with the viewing direction vector. The cosine of the angle between the viewing direction and each object – space vector axis is proportional to the dot product of the viewing direction vector with each object space unit vectors. The largest dot product corresponds to the smallest angle. From the Equations 7 and 8 the permutation matrix of the each viewing axis can be derived.

If the principal viewing axis is x0, then the standard object coordinates (*i, j, k*) correspond to the object coordinates (*y0, z0, x0*), and the permutation matrix is:

$$P(x0) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ **Equation 9**

If the principal viewing axis is y0, then the standard object coordinates (*i, j, k*) correspond to the object coordinates (*z0, x0, y0*), and the permutation matrix is:

$$P(y0) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ **Equation 10**

For the principal viewing axis z0, the standard object coordinates (*i, j, k*) correspond to the object coordinates (*x0, y0, z0*), and the permutation matrix is:

$$P(z0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ **Equation 11**

For parallel projections:

Considering $M'view = Mview * P^{-1}$ **Equation 12**

Then $$Mshear = \begin{bmatrix} 1 & 0 & Si & Ti \\ 0 & 1 & Sj & Tj \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ **Equation 13**

where $Si = \dfrac{M'22M'13 - M'12M'23}{M'11M'22 - M'21M'12}$ and $Sj = \dfrac{M'11M'23 - M'21M'13}{M'11M'22 - M'21M'12}$

$$Mwarp2D = \begin{bmatrix} M'11 & M'12 & M'x \\ M'21 & M'22 & M'y \\ 0 & 0 & 1 \end{bmatrix}$$ **Equation 14**

where $M'x = M'14 - TiM'11 - TjM'12$ and $M'y = M'24 - TiM'21 - TjM'22$

Translation coefficients Ti and Tj specify the displacement from origin of the standard object coordinate to intermediate, and are defined by Si and Sj signs.

Figure 10 shows the intermediate image coordinate system.

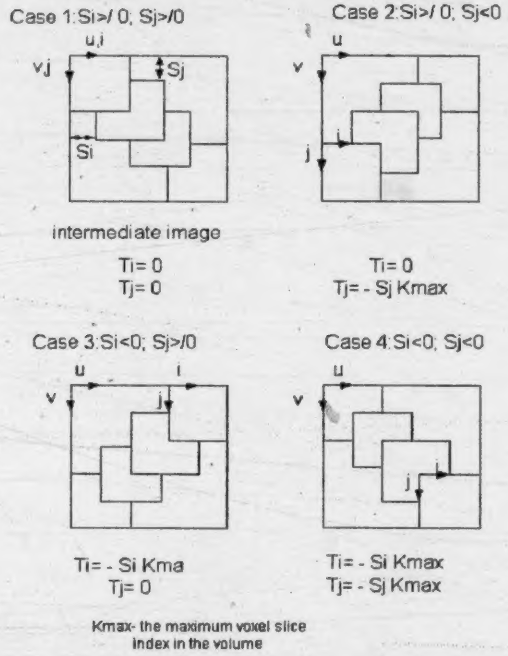


Figure 10 Intermediate image coordinate system

Therefore, to compute the parallel factorization the following steps are performed:

- Find the principal viewing axis and choose P matrix
- Define M'view matrix
- Define S(i,j) coefficients and T(i,j)
- Compute Mshear and Mwarp

In perspective projections the viewing rays are not parallel, so it is may not be possible to choose a single consistent principal axis. A perspective viewing transformation transforms the eye object space e_0 location to the image space point with homogeneous coordinates (0,0,-1,0). Therefore, the location of the eye can be computed by solving of the next:

$$\begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix} = Mview * \begin{bmatrix} e_{0,x} \\ e_{0.y} \\ e_{0,z} \\ e_{0,w} \end{bmatrix}$$ **Equation 15**

The next step forms vectors from the eye-point to each corner of the volume and defines the principal viewing direction for each of the eight directions that forms corners of the box for all viewing rays, which hit the volume. The largest axis projection, corresponding to the vector from the eye to the corner is the principal viewing axis for the particular viewing ray. This procedure is repeated for the each corner of the volume, and if the eye point coincides with any corner, then that corner should be omitted. In the case when the principal axis is not the same for the all eight corners, so the volume must by subdivided in the classification stage, and then proceeded for each of the defined principal viewing direction. Permutation matrices for the perspective projections are the same as for the parallel projections.

For perspective projections transformation from object coordinate to standard object coordinate is the same as for parallel projections. M'view transforms voxels from standard object coordinate to image space:

$$M'view = Mview * P^{-1} * T^{-1}shift$$ **Equation 16**

Considering that voxel slice has a scale factor of 1, we obtain:

$$M'shear = \begin{bmatrix} 1 & 0 & -\dfrac{e_{0,x}}{e_{0,z}} & 0 \\ 0 & 1 & -\dfrac{e_{0,y}}{e_{0,z}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\dfrac{e_{0,w}}{e_{0,z}} & 1 \end{bmatrix}$$ **Equation 17**

However, the slice is not necessarily the front voxel slice. The transformation scaling matrix is:

$$S = \begin{bmatrix} f' & 0 & 0 & 0 \\ 0 & f' & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ **Equation 18**

where $f' = \begin{cases} 1 & if\_slice\_k = 0\_is\_in\_front \\ 1 - \dfrac{e_{0,w}}{e_{0,z}} \times k_{\max} & otherwise \end{cases}$ **Equation 19**

According to the Figure 10 the translation point $T(t_i, t_j, 0)$ positions the upper-left corner of the bounding box at the origin. Then, the complete shear factor is:

$$Mshear = T \times S \times \begin{bmatrix} 1 & 0 & \dfrac{e_{0,x}}{e_{0,z}} & 0 \\ 0 & 1 & \dfrac{e_{0,y}}{e_{0,z}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \dfrac{e_{0,w}}{e_{0,z}} & 1 \end{bmatrix}$$ **Equation 20**

The warp factor is given by:

$$Mwarp = M'view * M^{-1}shear \times S^{-1} \times T^{-1}$$ **Equation 21**

To compute the perspective factorization the following steps should be performed:

- Find the location of the eye in object space
- Find the principal viewing axis and choose P matrix
- Determine Tshift and compute the permuted view matrix from Mview and P
- Compute the shear and perspective scale coefficients from the eye coordinates
- Computer the scale and translation that complete the transformation to intermediate image coordinates
- Compute the shear matrix Mshear and the warp matrix Mwarp.

To summarize the all of the above: the matrix multiplication processor becomes the cornerstone of the shear-warp factorization (SWF) VRS. Chapter 3 describes the designed Matrix Multiplication Processor (MMP) based on the inner product operation.

1.1.3.5. Shear-Warp Factorization Rendered Examples

This section concludes the introduction to the shear-warp factorization algorithm shown in Figure 11 [5].

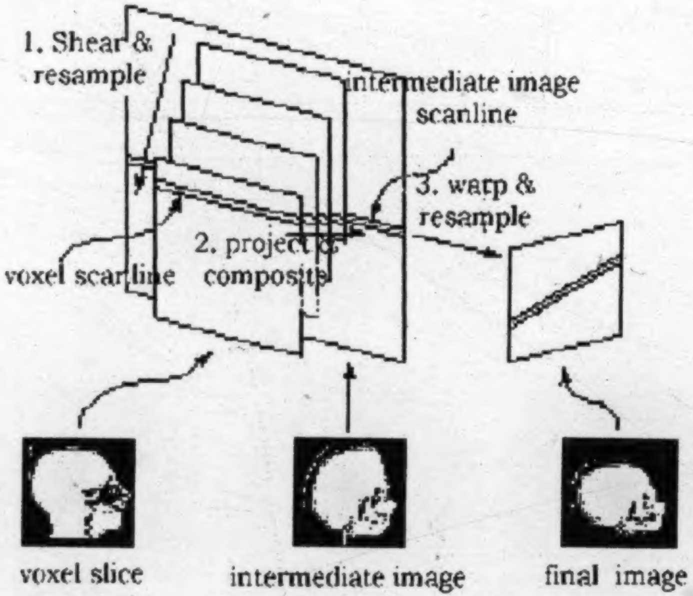


Figure 11 Shear - warp factorization algorithm

The image in Figure 12 was rendered using Volpack software developed by Lacroute and described in [4]. Volpack is an implementation of the shear- warp algorithm.

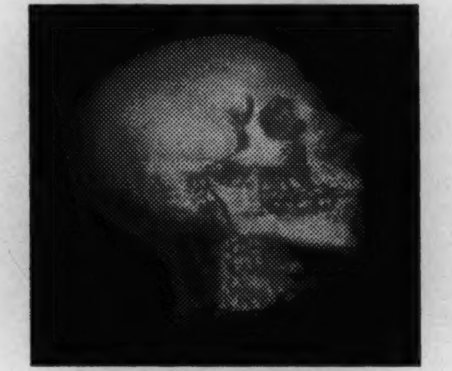


Figure 12 Human head image rendered using shear-warp factorization algorithm

The image in Figure 12 was obtained from the following dataset:

Description: Magnetic resonance study of human head 226x256x256 slices.
Dimensions: z-226 y-256 x-256
Order: z-y-x
Format: 8 bits scaled down from the original 16 bits.
Data source: Data taken on the Siemens Magnetom and provided courtesy of Siemens Medical Systems, Inc., Iselin, NJ.

Rendering header information is presented below:

magic:	192837466	scale X:	1.000000
header length:	60	scale Y:	1.000000
width:	256	scale Z:	1.100000
height:	256	rotate X:	0.000000
images:	226	rotate Y:	0.000000
bits/voxel:	8	rotate Z:	0.000000
index bits:	0		

The illustration of two consecutive slices is shown in Figure 13 [5]. These image slices were rendered from the same data set as the previous image found in Figure 12.

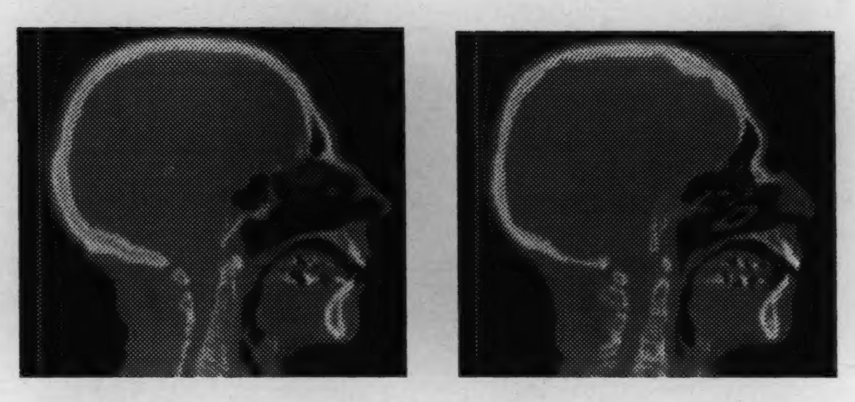
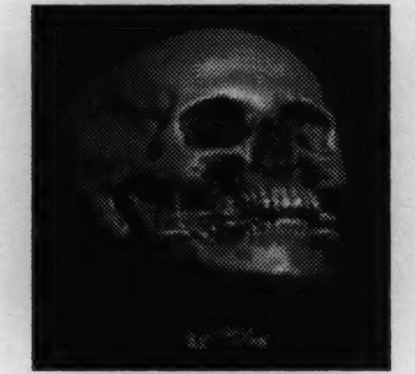


Figure 13 Volume consistent slices in z-axis direction

The classification transfer functions in Figure 14 have been changed to make soft tissues completely transparent, so only the bone surfaces are visible. A rotation transformation matrix Mrot has been applied to with rotation angle α.



$$Mrot = \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 14 Classification example of the rendered image

Phong [38] fast lighting model has been used to shade the image in Figure 12.

Figure 15 illustrates how changes in the gradient of the volume data affect three-dimensional structure of the data.

All these rendered examples visually introduce the function of each of the components of the volume rendering pipeline developed in SWF VRS, such as: gradient processor, matrix multiplication processor, shader and classifier.

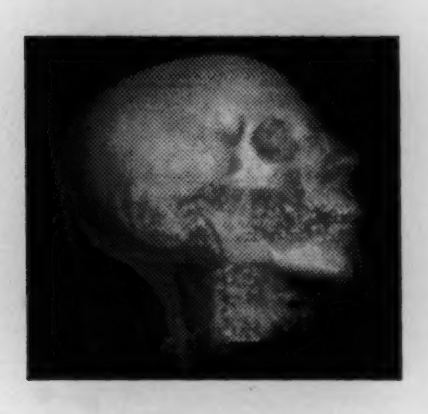


Figure 15 Lightened model

1.1.4. Thesis Organization

Chapter 2 begins with a brief survey of volume rendering implementations. A number of software solutions, methods implemented, accelerating techniques and its advantages and disadvantages, are presented. The much fewer hardware implementations are the real research interest of the following chapter. The existing and the most recently declared real – time volume visualization systems are presented and discussed; the primary limitations of those VRS, and how the proposed VRS improves these drawbacks, or might be just generally free of them because of a different approach.

The designed VRS has four levels of hierarchy: system level, component level, circuit level and logic level. Chapter 3 describes the system level design with structural and functional diagrams and a novel approach, which provides coherent interactive memory access. It shows the design data flow from logic to architectural level.

Chapter 4 introduces components level design of the VRS. The following components are described: Matrix Multiplication Processor, Sobel Operator Processor, Shear-Warp Factorization Composer, Shader and Classifier.

Chapter 5 contains the circuit design descriptions. The term "circuit" refers to subcomponents, like inner product processor, encoder, adder, compressor, etc. These relatively small functional subcomponents, in turn, form components, described in the previous chapter. Subcomponents' design, verification, manufacturing and testing issues are also explained in this chapter. Also Chapter 5 describes logic level design and verification. A low power and high-speed transistor logic library based on reduce swing CPL is presented.

Finally, Chapter 6 discusses the implications for future research and gives the conclusions of this work.

Chapter 2

2. Volume Rendering - State of the Art

2.1. Volume Rendering Systems

The first implementation of a volume rendering technique grew out of research done at the Mayo Clinic in the 1970s. In the mid-1980s, rapid growing of demands on image processing hardware and integration of new data manipulation techniques initiated the research in volume rendering. Volume rendering has been implemented in both: specialized hardware and software for more readily available systems.

2.1.1. Software Implementations

The following list gives the major software implementations:

Splatting: ParVox [80], AVS [81], Pixar [82], VolVis [83].

Shear- Warp Factorization: VolPack [6] [4], VolRend [84], VolSh [85].

Ray- casting: UltraVis [86], 3D-Doctor [87], OpenQVIS [91], XRAY, KROT [88].

Others: MPIRE [89] (Splatting, Ray casting).

Image and volume rendering packages and plug-ins, which use non-direct VR algorithms and run on different platforms: VoxBlast [90] (network, measurements oriented), BOB [92] (GVLware), 3DviewNix [93], VFleet [94], Dr.Razz [95], VRVis [96] (Virtual reality), KHOROS [97][78], OSIRIS [98][79] (medical imaging), SIMIAN [99].

Software implementation advantage and disadvantage:

Advantage: Software implementations eventual and obvious advantage lies in the loose bounding to the platform they are written for. This allows a certain level of platform portability that is not possible with hardware implementations.

Disadvantage: The main downside of software implementations is limited rendering speed. A machine-independent approach puts more demands on the central processing unit (CPU) and cannot take advantage of specialized chip sets that are available.

Parallel computers, consisting of tens of processors, have been used in the past few years in an effort to improve the rendering speeds of volume rendering algorithms [2] [63] [67]. Parallel processing dramatically improves rendering speed by computing all of the pixel values in a 3D image in parallel rather than serially as is typically with software implementations. 3D images can be rendered in real time at rates exceeding 5–10 frames/sec [64].

Frame rates for 128x128x109 volume buffer renderings of 31Hz have been reported by Lacroute [3] from a 16-processor machine (SGI Challenge) using the shear-warp algorithm. The same algorithm produces 13Hz for a 256x256x225 buffer on the same machine, however, that the pre-computation time on a single processor, data voxelisation, was 65 seconds [5]. Thus, real-time visualization is not reachable, and this leads to the requirements for the hardware solutions.

2.1.2. Hardware Implementations

There are very few hardware implementations of Volume Rendering systems [71]. Several researchers have proposed special-purpose volume rendering architectures [9]. VOGUE and VIRIM are ray-casting architectures. VOGUE [13], a modular add-on accelerator achieves 2.5 frames per second for 256^3 datasets. VOGUE required 64 boards and a 5.2 GB/sec ring-connected cubic network to achieve 20 frames per second of 512^3datasets. VIRIM [5], a programmable ray-casting engine, requires duplication of data and 16 boards for rendering 128x 256x 256 datasets at 10 frames per second.

A programmable co-processor architecture for volume processing [70], called PAVLOV (Parallel Array for VoLume prOcessing and Viewing), was recently developed. The system is capable of rendering volumes at 30Hz frame rates and performing volume processing tasks at interactive to real-time rates. The advantage of the PAVLOV system is that it is programmable. Another alternative method is used to process the dataset to measure the frequency of the underlying pattern in a small neighborhood of voxels. These values, which represent the density frequency in the structure of the underlying data, are then rendered into a 2D image to be presented to the user.

2.1.2.1. CUBE Project

One of the earliest hardware implementation and the most significant was started in 1991 at SUNY Stone Brook University, called the CUBE-1 project [12], which has long implementations history from Teramac [55] to FPGA [56] and to ASIC [44].

The latest one, as a proof of concept, CUBE-4 was implemented in HP TERAMAC [55], a configurable custom hardware machine developed at Hewlett-Packard Laboratories, using the ray-casting algorithm.

Teramac can execute synchronous logic designs of up to one million gates at rates up to 1 MHz. The system has been built from custom FPGAs packaged in large multichip modules (MCMs). The Teramac system for Cube-4 implementation includes 8 boards, 250 MB of RAM, 32 MCMs and 864 FPGAs.

The implementation is capable of producing parallel color projections of 8-bit per voxel datasets from arbitrary directions. The total logic complexity for all five rendering pipelines is 330K gates. The Cube-4 prototype generated an image of the datasets in 1.5 seconds at 0.25 MHz, independent of dataset.

To facilitate a compact implementation, an ASIC containing several of the Cube-4 rendering pipelines was developed. Its preliminary estimation was that an ASIC containing 4 rendering pipelines required less than 300 pins, including power and ground. Each ASIC required only 400,000 gates and internal memory for buffers of 40 KBytes.

Original plans were made in 1997, and in 1999 the VolumePro[44], described below, the first and still the only one ASIC for Volume Rendering was delivered by Tera Recon, a sub-division of the Mitsubishi Corporation.

It is a hardware implementation of ray casting using the shear-warp algorithm approach of the baseplane image rendering. The 2D warp operation must be done by common graphics hardware. It provides real-time rendering with compositing, classification with density based transfer functions and Phong shading.

VolumePro is the rendering kernel in several rendering architectures. One of the examples is the Sepia architecture [69] targeted for the interactive rendering for scalable volumetric data, where the VolumePro 500 was used as a ray-casting engine.

2.1.2.2. VIZARD II

VIZARD II is the most recent first fully FPGA-based implementation of a ray-casting pipeline. It is the result of a collaboration of the University of Tubingen and Stony Brook University [58]. It is implemented on the Vertex FPGA, the architecture of this PCI board is shown in Figure 16, and ray-casting engine is shown in Figure 17.

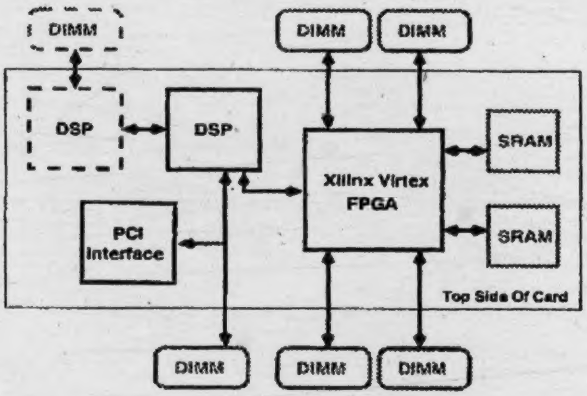


Figure 16 Architecture of VIZARD II PCI board

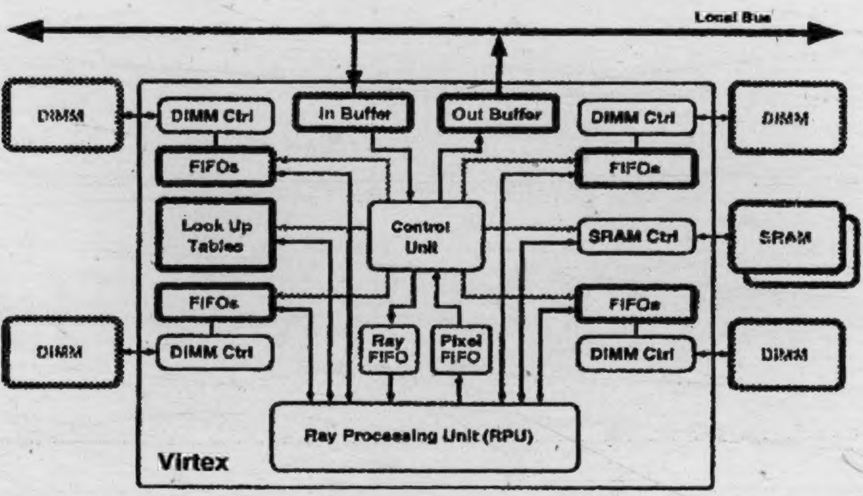


Figure 17 VIZARD II ray-casting engine

VIZARD II interactively processes a 512^3 volume dataset. However, due to classification and volume size dependency, frame rate is not constant, and thus a real-time rate is not constantly provided.

2.1.2.3. VolumePro

TheVolumePro system [44] is based on the Cube-4 volume rendering architecture developed at SUNY Stony Brook. Mitsubishi Electric licensed the Cube-4 technology and developed the Enhanced Memory Cube-4 (EM-Cube) architecture.

It's highly parallel architecture based on the object-order ray-casting algorithm, described previously. The second upgraded chip generation called VolumePro 1000 was released in 2001. The chip was built using IBM's SA27 0.16u process, boosting pipeline speed to 250MHz from 0.35u of the very first rendering chip-VolumePro 500. The 216-mm 2-die core includes 2.3 million gates of logic plus two million bits of SRAM, requiring 1247-contact ceramic BGA package. The price of this product ranges from 3000$US (board alone) to 5500$US (with software package). Improvements to this chip include some accelerating techniques, which are considered as optional. As a result the VolumePro 1000 is able to render a 512x512x512 voxels image in the real-time mode, 30 frames per second. Architecture of VolumePro is given in Figure 18.

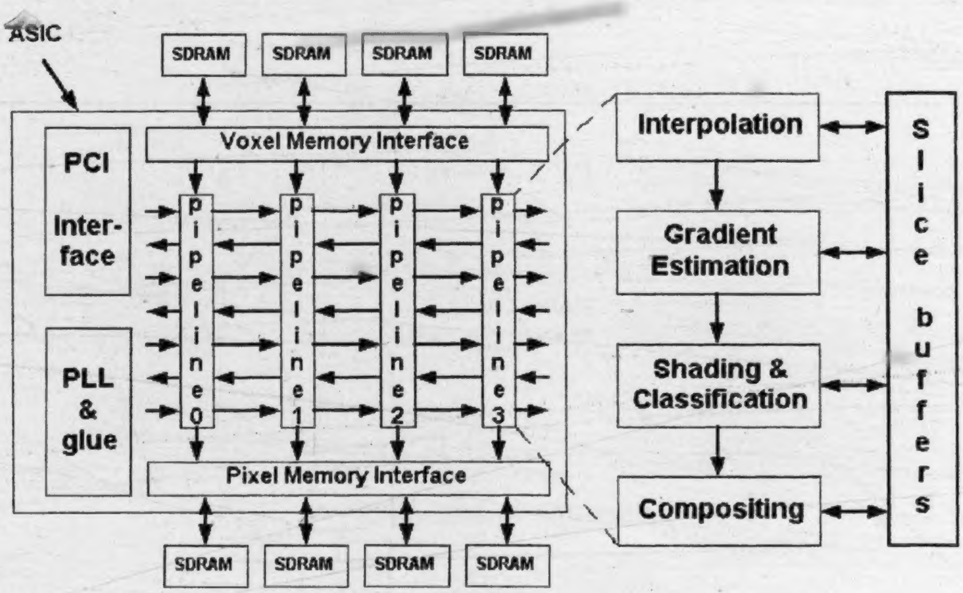


Figure 18 VolumePro architecture

Detailed the VolumePro™ 1000 Key features are given below:

- Real time performance for volumes up to 512 x 512 x 512,

- Stores between 256MB and 2Gbyte of volume memory on a single PCI board,
- Ability to embed opaque and translucent polygon surfaces within the volume data,
- Concurrent volume updating and rendering,
- Classification and interpolation in either order,
- Up to 8K voxels in any dimension in one pass,
- Ignores voxels not contributing to visible sample for improved performance,
- Depth and image filtering based on opacity, gradient direction, and gradient magnitude,
- Gradient lighting applied to every sample point to make samples look like surfaces.

However, VolumePro has several drawbacks, which limit its usage and image quality or even, in rare cases, render with visual artifacts.

- One of the main drawbacks of the VolumePro board, with regard to usage, is that it does not produce perspective projection. For outside views, parallel projection gives good results but for inside views, perspective projection is mandatory to provide a correct depth impression.
- Another significant drawback is that currently it is only able to perform back-to-front order compositing.
- For the VolumePro only directional light sources can be defined. In some cases it is quite convenient to use a head-mounted point light source.
- One of the drawbacks in sliced-based methods is that insufficient interpolation leads to severe visual artifacts. The VolumePro board tries to solve this problem by several levels of super sampling. The disadvantage is that super sampling results in a corresponding slow down in rendering performance.

2.1.3. Chapter Summary

This chapter presented existing VRS implementations either in software or hardware. However, emphasis has been given to volume rendering hardware implementations.

The following Chapter 3 describes the architecture of the designed VRS, including BIST aspects, explains the advantages of the time-slicing approach, gives the design flow, and defines the functional load of the each component.

The SWF VRS improves the limitations of previous system described above in the following way:

- It produces perspective projections.
- Compositing is done in either way: front –to-back or back-to-front.
- Front clipping solves this problem: the user defines the distance from the viewpoint to the first slab.
- The super sampling is implemented in such a way that the slabs closer to the viewpoint are rendered with a higher super sampling level, in other words, differential level of super sampling for the each slab.

Chapter 3

3. Shear – Warp Factorization Volume Rendering System

3.1. System Design

3.1.1. Design Flow

The purpose of the current work is to provide real-time volume rendering on a PC based platform. The volume rendering system probably still is considered as a special purpose device to be implemented as a plug-in card with an interactive users' interface.

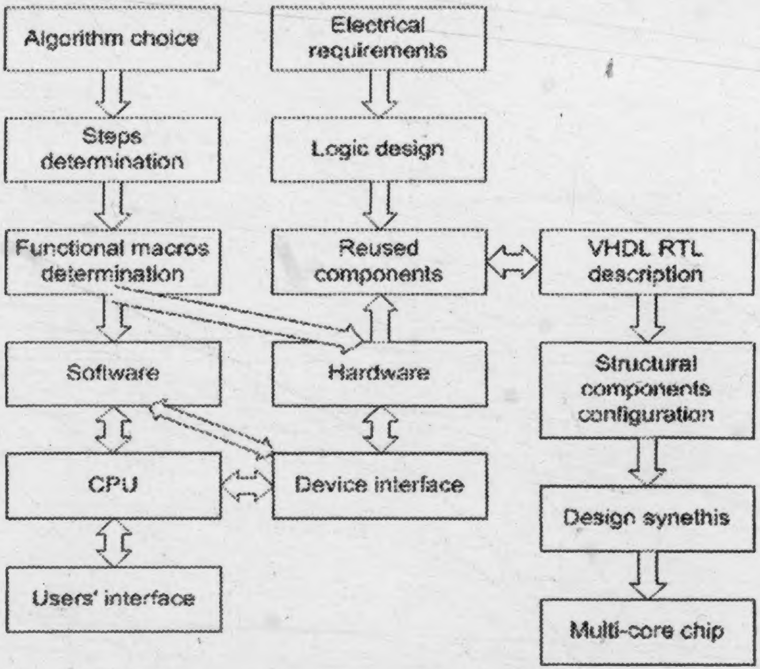


Figure 19 Design development data flow

The difficulties to be considered involve multifunctional blocks, complex architecture and complex layout. Our approach solves these problems.

Figure 19 shows the design development flow. The correct functional division onto either subsystems or simply functional modules will provide the optimal architecture. The aforementioned rendering pipeline in Figure 2 dictates these functional blocks.

Since segmentation is considered as a pre-processing step, and due to its extreme complexity and algorithm dependency requires the intervention of a human, it should be implemented in software.

Classification is a process that can be completely automated by assigning opacity values and colors to voxels according to the user interest, for example, to set up the labeled 'bone" voxels as color gray, the "blood" voxels as color red, or the color green to "tissue" voxels; it requires a user's interference and a large amount of memory to store look-up tables with pre-calculated opacity values. Therefore, a software implementation will be optimal for this module as well.

The other modules, the gradient calculator, the shearer-composer, the shading processor and the matrix multiplication processor are composed from the basic functional blocks such as: multipliers, adders, or inner product processors. Implemented as independent blocks, these components form the design hardware "library", where components will be reused. Moreover, these functional blocks physically are considered as sub-cores; based on the type of processing required by the calling module, the embedded sub-core is configured or replicated to execute a specific operation. A variety of functions can be executed using a single component, thus resulting in embedded components that can produce more efficient systems.

Using this approach design hierarchy will significantly simplify layout. On the other hand, multi-core layout with several independent vertical cores allows the user to choose a required function, and, therefore, utilize plug-in devices not only for the particular shear-warp factorization algorithm, but also for core standard applications for the volume rendering using any rendering algorithm.

Another requirement, since the VRS might be considered to be used with any portable PC as well, low power consumption will be its significant asset. To implement the SWF VSR the following steps are considered in the design:

- Volume rendering algorithm chosen
- Standard cores from rendering pipeline designed
- Specific cores for the chosen algorithm designed
- Component reuse library designed

- Low power low voltage logic library designed
- Components configured hierarchically into cores, and cores into system
- Pipelining stages considered
- Parallel processing considered
- Users' interactive interface built
- Timing requirements to provide real-time mode
- Bus inputs sampling
- Bus output re-timing
- CPU communications including interrupts generation
- Memory requirements

Verification

- RTL Testbenches
- Scripts to compile, analyze and elaborate code
- Timing analysis

Design synthesis

- VHDL code synthesis ability
- Synthesis script

Testability

- BIST
- Scan insertion

Layout

- Automatic layout generation
- Custom layout
- Verification
- Timing analysis

Chip manufacturing

Chip testing

Plug-in card manufacturing

The current theses completed job includes the design on the all hierarchical levels, implementation, verification, and partly custom layout. The SWF system provides the real-time rendering due a time slicing approach, and the implementing of a multi-core structure complied with the rendering pipeline.

3.1.2. VRS Time-Slicing Architecture

The shear- warp factorization volume rendering system, shown in Figure 20, contains the following blocks, mentioned in the volume rendering pipeline in Figure 2: classifier, shader, gradient calculator, and composer.

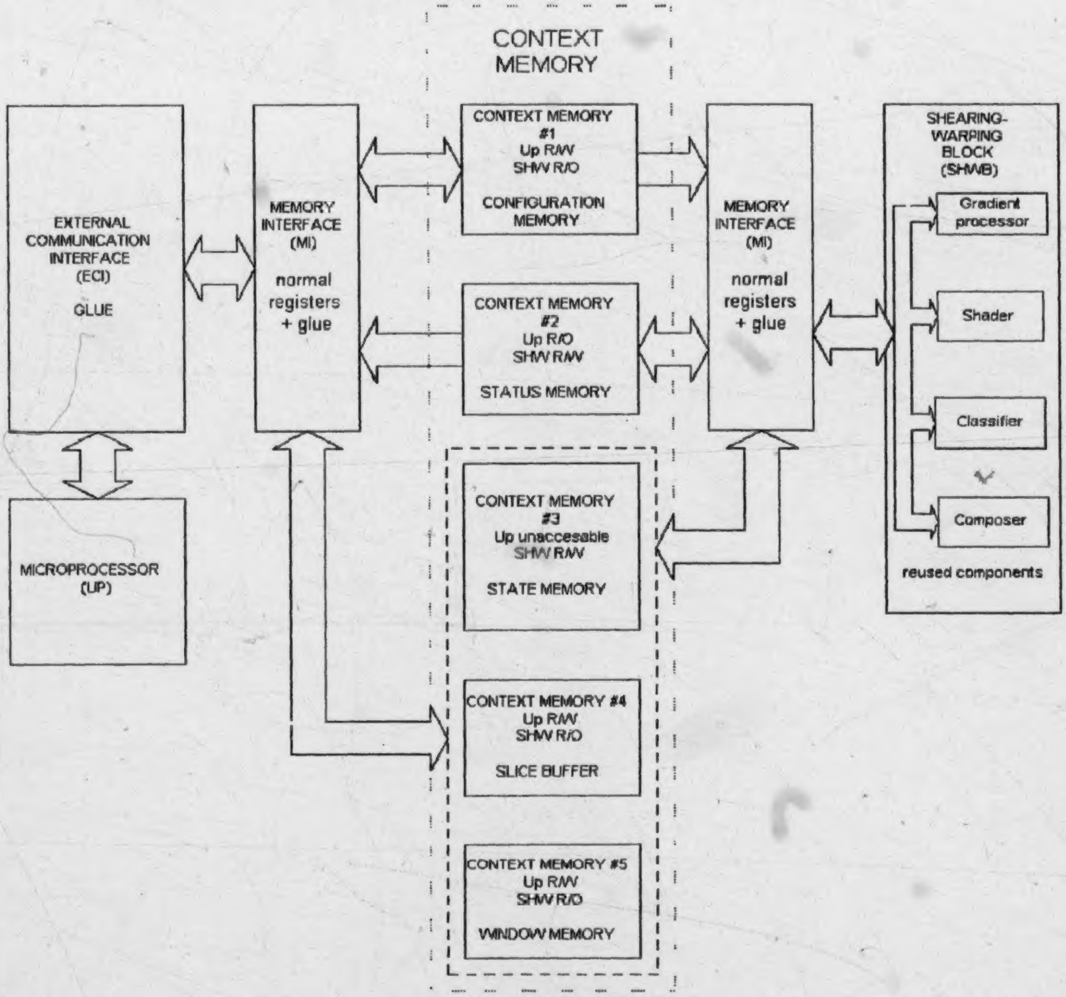


Figure 20 SWF VRS structure

The system is built using component reusability as the main principle. Time-sliced pipelined architecture and component reusability allow simultaneous processing of several tasks. Moreover, this approach provides integration and continuous communication of "hardware" components with "software" components, and the interactive user's interface.

For example, the calculated dot product of the incoming voxel is used in the shader and the composer. Computed once, it is stored for several clock cycles in the corresponding RAM cell to be called up by another component.

To communicate in such a way, components must have a common interface provided by context RAM, normal mode registers and a context address snooping mechanism, where stored information is assessable for reuse. Each component has its own context RAM, although physically it is combined into the common memory. The memory type (either flip flops (FFs) or any RAM) and size is defined by its destination and efficiency considerations.

Registers are used to configure and monitor the operation modes of the VRS. All the interrupts, configuration and status vectors stored in the corresponding context RAM are accessible through these registers. They might be configured for two working modes: normal mode operation and testing mode operation. The external communication interface (ECI), which physically might be thought just as "glue", provides dockability of the system "hardware" part/components to "software" part/components.

Although registers and the ECI might be combined into one block, which still would be called the External Communication Interface block, and, which plays a very important role of the communication channel between user or/and microprocessor and hardware components of volume rendering system, in our design we separate them into two different components.

3.1.2.1. External Communication Interface

Top level ECI structure is shown in Figure 21.

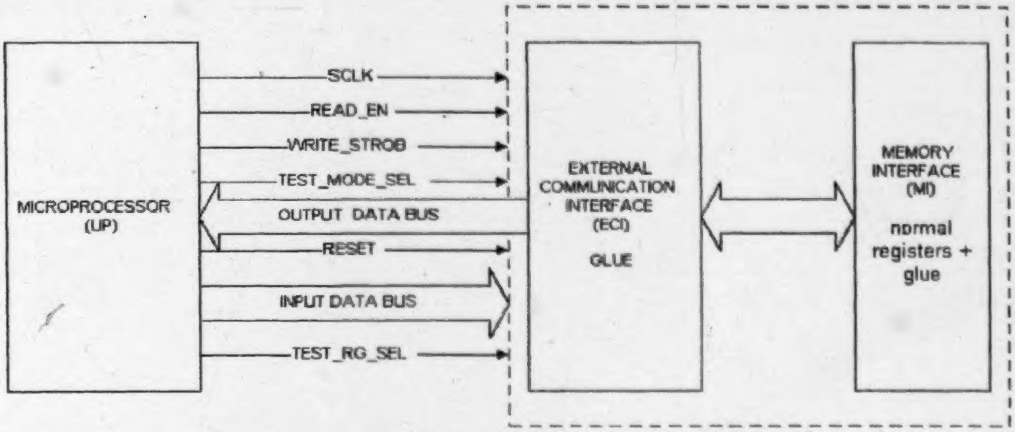


Figure 21 Communication interface structure

The registers and ECI logic contain configuration, status and state information for the each component and the entire system. This information is stored in Context RAM and is accessible through a set of indirect access registers within ECI address space, and a number of status and state registers.

The number and indirect registers' contents will be finally defined upon the building of the entire system as well as a system test mode. However, the each component has its own Context RAM, hence, corresponding ECI/registers.

Indirect access address is defined by the following considerations: the maximum rendering image sizes is assumed to be 512x512x512; to compute gradient, opacity or transform computations window pixel information and slice information is required. This will lead to the following formula to compute the unique context RAM address:

Context RAM address = Slice number + (rang from 0 to N-1)
+ pixel number horizontal row + (rang from 0 to N-1)
+ pixel number vertical row, (rang from 0 to N-1)

Figure 22 illustrates this formula, context RAM snooping calculations, and how the pixel window is formed.

In addition to specifying the particular voxel (pixel) in the context RAM, the indirect address register contains bits, which identify whether an access to read or write, and busy bit indicating that a transaction is currently in progress. Then, a complete address value is written to the indirect address register.

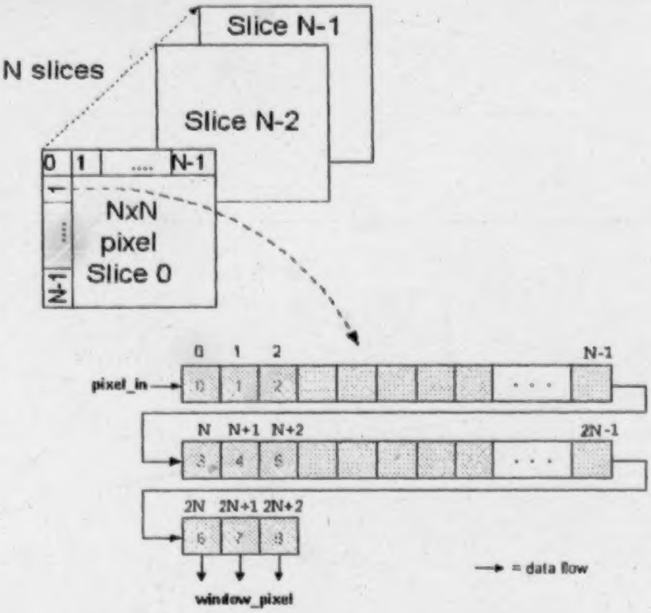


Figure 22 Context RAM snooping address formation

3.1.2.2. Context RAM

The function of the Context RAM is to store the states of the time – sliced state machine as computed by a component or components, component configuration vectors and component status information. Context RAM consists of three functionally separate RAMs shown in Table 3.

Table 3 Context RAM

	Bit	Name	Description
1	Start of microprocessor accessible read/write Configuration RAM		
	0		
	…		
	Config_wd		
	End of microprocessor accessible read/write Configuration RAM		

2	Start of microprocessor accessible read –only Status RAM		
	0		
	…		
	Status_wd		
	End of microprocessor accessible read –only Status RAM		
3	Start of microprocessor inaccessible State RAM		
	0		
	…		
	State_wd		
	End of microprocessor inaccessible State RAM		

Context RAM is a dual port memory device that depending on its destination can be implemented in flip-flops, ECC RAM or any other type of memory; efficiency will determine the kind of memory choice.

In the Configuration RAM (1) one port is dedicated to VRS reads, the other to microprocessor (UP) writes. In the Status RAM (2) one port is used by state machine /machines to write back new status vectors of VRS, and the other to reads by microprocessor. In the State RAM (3) time slice machine can only write to one port and read the other. Should be pointed out that the State RAM may or may not exist for a component.

If the new status information to be written back has not changed, and is equal to the old, there is no write back operation, and the previous data is kept. Then signal indicating status change (*new_info*) is set low. In this case dual port (dpr) memory has a considerable advantage- microprocessor's reads can use an idle cycle, providing almost instantaneous access. However, state bits are always written back.

Some external "glue" and a single MUX on the Status RAM on the address in port 1 are required to create the entire Context RAM interconnections picture.

Context RAM is generated using the generic RAM model given in the Appendix1. Figure 23 shows the Context RAM ports.

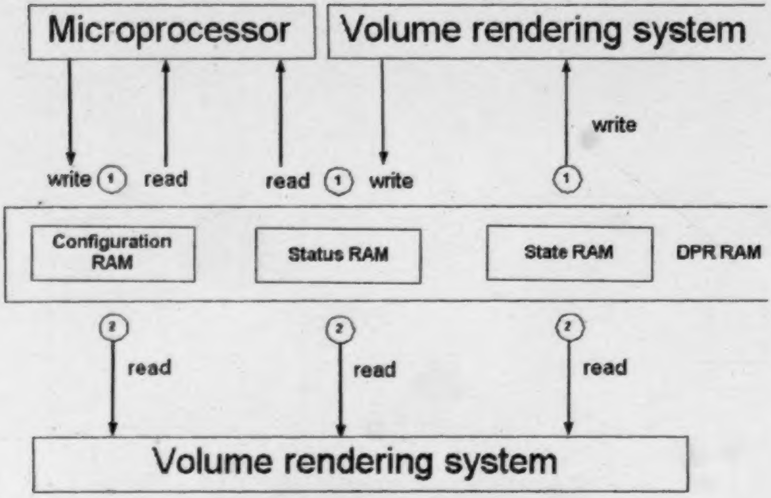


Figure 23 Dual port Context RAM ports

Since we consider indirect access to the Context RAM the problem with simultaneous read and write access for the Configuration RAM might occur. The state machine shown in Figure 24 avoids this contention. The state machine will stay in the read state until no write back cycle occurs.

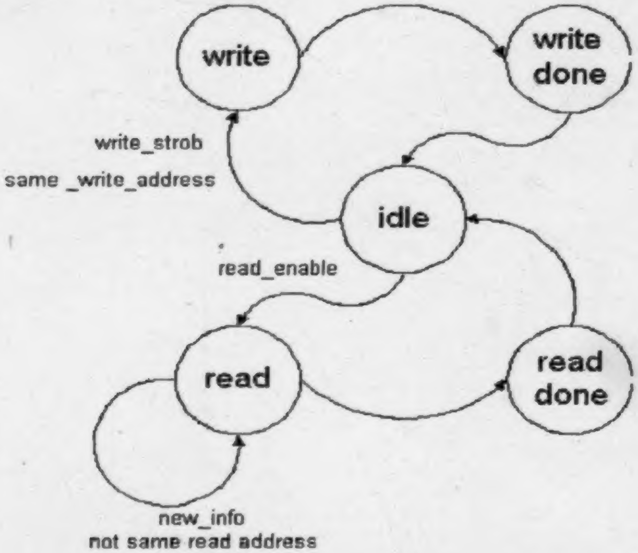


Figure 24 Microprocessor to Context RAM access state machine

3.1.2.3. Testability

Achieving low power consumption increases circuitry reliability, however, to conduct a functional test, testability aspects like Built-in-self-test (BIST) must be considered. Design for testability (DFT) is provided by inserting extra I/O pins besides the functional pins. The port *Test* is such an extra pin (Figure 25).

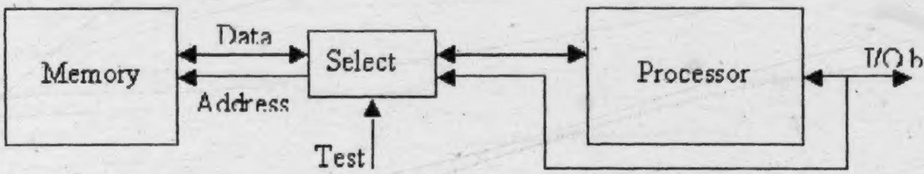


Figure 25 Inserting testability by inserting multiplexers

To reduce the number of extra pads one can multiplex test signal and functional signals on the same pads. This AD HOC approach is combined with embedded BIST architecture. Hardware synthesis techniques automatically generate a structural hardware implementation given an abstract description of the behavior of the part. BIST techniques are implemented by including a *test pattern generator (TPG)* and *output response analyzer (ORA)* in the part. During the BIST operation, the TPG applies test patterns to the inputs of the circuit under test (CUT), and the ORA captures the response of the CUT to those test patterns.

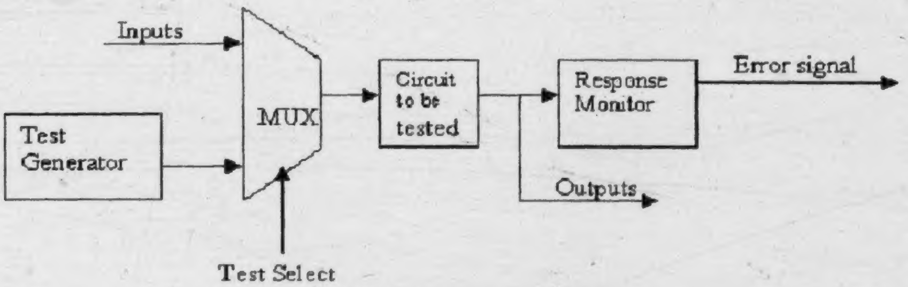


Figure 26 Embedded BIST

Embedded BIST architectures, shown in Figure 26, use re-configurable system bi-stables to implement both the normal system operation and the BIST TPG and ORA operations. Since the BIST logic is combined with the system logic, opportunities exist for synthesis techniques to generate hardware that can be shared by both the system and test operations, resulting in improved performance and reduced cost.

Test benches emulate a hardware breadboard into a synthesizable design, which description will be "installed" for the purpose of verification. Depending on our needs we develop one or more test benches to verify the design functionally (with no delays), to check our assumptions about timing relationships (using estimates or unit delays), or to simulate with annotated post-route timing information so we can verify that our circuit will operate in-system at speed. During simulation, the test bench will be the top level of a design hierarchy. Our approach lies in creating test stimuli to describe the test bench in terms of a sequence of fixed input and expected output values, as shown in Figure 27.

BIST considers comparison of known final outputs with those obtained. For an extended algorithm, like any rendering algorithms, several "break points" might be used. Pre-computed step-by-step data set is used with known responses in pipelined modules. It does not necessarily require much additional hardware. An existing hardware will be used, however, additional external test signals are needed.

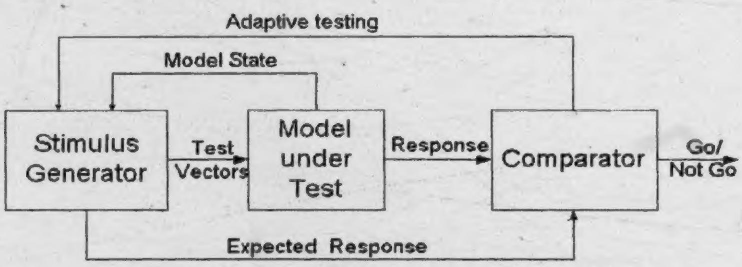


Figure 27 BIST using test benches

Particularly for the shear – warp implementation, where parallelism and pipelining are commonly used, pipelining registers were used as "break points" when the *Mode Sel* signal chooses the mode: either normal operational mode, or activate test route and start testing cycle. A structural schematic of the SWF BIST is shown in Figure 28.

The test pattern generator can be changed by user through context RAM since data from both, generator and response analyzer sides, are delivered by normal registers.

The big advantage of the proposed approach is in flexibility of use. We can test any size of CUT. An example might be the entire Wallace tree or each of its compressing stage.

The only question is in optimal distribution of inserted scans within the design for the optimal circuit testing.

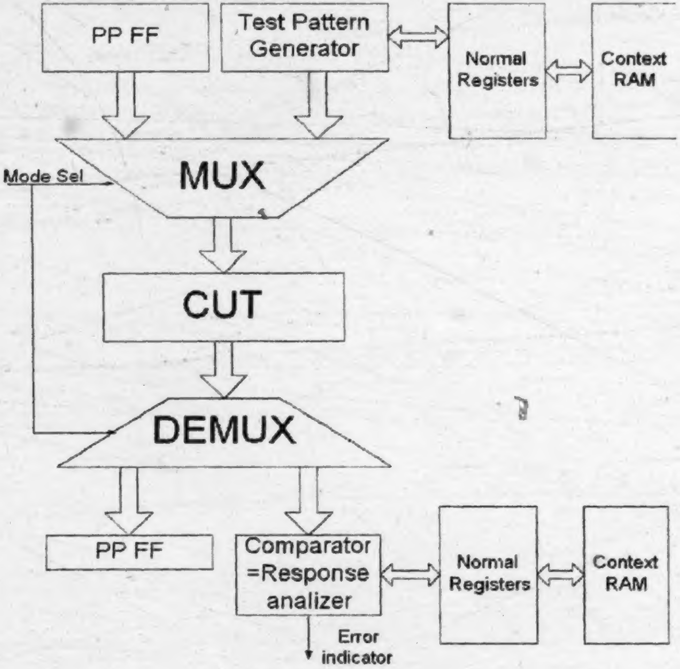


Figure 28 SWF VRS BIST

3.1.2.4. Pipeline Stages

Pipelining is one of the major advantages of the time slicing architecture. Pipelining brings to this design the status of "high-speed design". Stages are combinational logic blocks, and pipe registers are just a simple set of sampling on the rising edge FFs.

Figure 29 illustrates the idea of pipelining used in the time slicing approach.

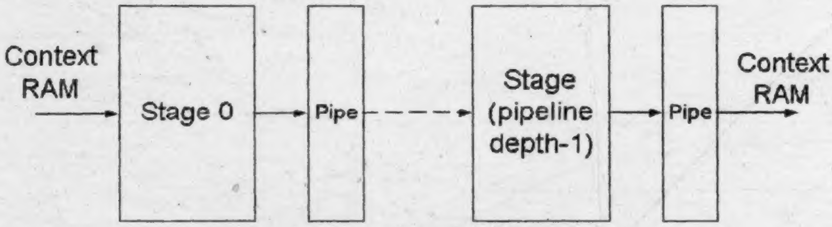


Figure 29 Pipelining stages structure

The pipeline depth is defined separately for the each component. However, since we consider computed data reuse, the functional timing should be taken under careful consideration and strictly verified by diversified testbenches. Chapter 5 gives detailed explanation of the pipelining issue.

3.1.2.4.1. Pipelining and clock tree

Pipelining stages must have the equal timing. If the system clock is not defined by system parameters, the most common approach is to divide a combinational circuit by convenient pipelining stages, and the biggest stage propagation delay should be less or equal the reference clock value.

However, to reduce the power consumption we can use several clocks that even might be different for pipeline stages that approximately equalize the value: clock (N) and N-stage propagation delay.

The main problem with the clock tree would be in clocks synchronization and metastability aspects that come together with multi-clocking. This will bring the necessity of FIFO controller embedding, and limit our choice of FF (since we need to fit the probability of failure by FF setup time, Tau parameter and FF delay).

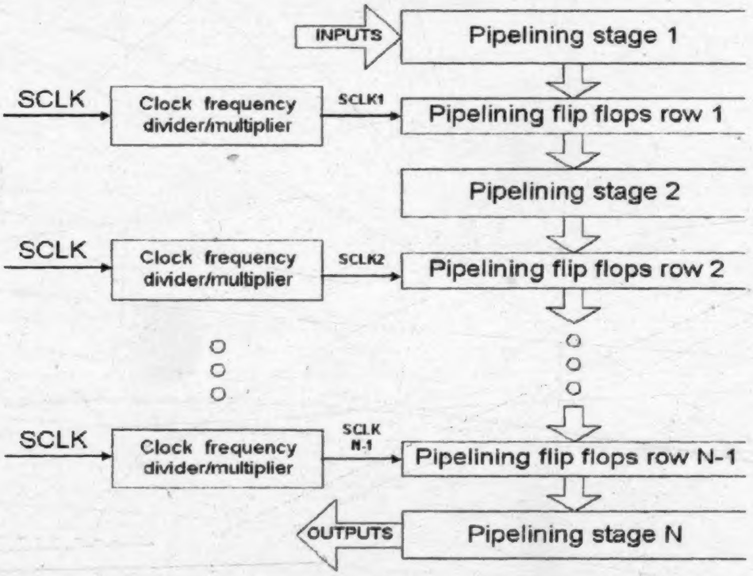

INPUTS
Pipelining stage 1
SCLK
Clock frequency divider/multiplier
SCLK1
Pipelining flip flops row 1
Pipelining stage 2
SCLK
Clock frequency divider/multiplier
SCLK2
Pipelining flip flops row 2
SCLK
Clock frequency divider/multiplier
SCLK N-1
Pipelining flip flops row N-1
OUTPUTS
Pipelining stage N

Figure 30 Pipelining and multi-clocking solution

The following approach, which solves these problems, still uses just one system clock, but before each pipelining registers array the programmable frequency divider/multiplier is inserted. Thus, we can program the value of the secondary clocks, which are already primary synchronized. This is structurally shown in Figure 30. Schematic of the clock frequency divider is shown in Figure 31.

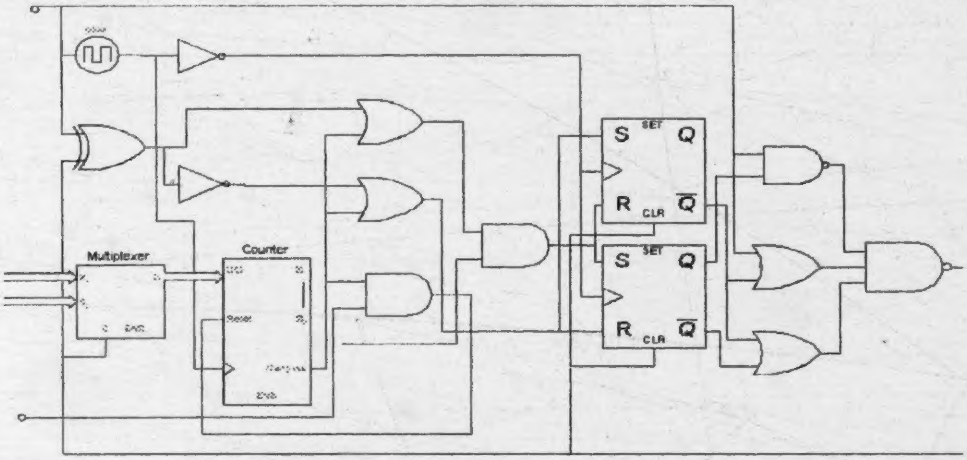


Figure 31 Clock frequency divider

Overall, the frequency of output pulses is equal to the input frequency divided by the coefficient defined by the half of the sum of the multiplexer inputs.

Detailed pipelining flip-flop design description is given in the Chapter 5.

3.1.3. Chapter Summary

In this chapter the first hierarchical level, the system level, of the SWF VRS design has been described including BIST aspects. We explained how components are composed into the system and how they are accessed in time slicing manner.

The following Chapter 4 describes the second design hierarchy level: component design. Volume rendering systems components vary in their place in the volume rendering pipeline, depending on the algorithm implemented, and in the way of their implementation. Brief review shows advantages and drawbacks of existing analogs.

Chapter 4 gives the mathematical basics of the shear-warp factorization algorithm and its architectural diagrams, and describes the components design and verification.

Chapter 4

4. SWF VRS Components Design

This chapter shows the design and verification of the rendering system components (sub-cores), a functional description of which was given in the Chapter 1. The following components are described in this chapter: the Gradient Processor, the Shearer-Composer, the basic component- Matrix Multiplication Processor, the Shader and Classifier. Although those components are combined into one chapter (since chapters are divided according to the system hierarchical levels), each of them is described separately in a chapter sub-title.

4.1. Gradient Processor

The gradient is a measure of how quickly voxel intensities in a data set change. It also tells the direction of changes. Therefore, the gradient carries the information about the structure of data set, and how quickly data changes in the data set.

4.1.1. Gradient Operators

The gradient can be calculated using several different methods. The gradient is used in two places in the volume rendering pipeline [76] [74]: in the shading stage and in the classification stage. Research has shown that accurate gradient estimation is one of the most important factors that determine the look and quality of a volume rendered image. Suppose we have the following signal, with an edge as highlighted in Figure 32.

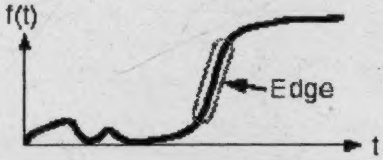


Figure 32 Approximated signal

If we take the gradient of this signal (which, in one dimension, is just the first derivative with respect to T) we get the following result, given in Figure 33:

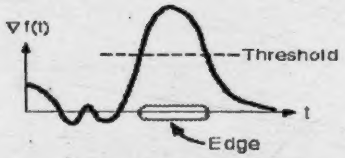


Figure 33 Gradient of the signal

Clearly, the gradient has a large peak centered on the edge. By comparing the gradient to a threshold, we can detect an edge whenever the threshold is exceeded (as shown above). In this case, we have found the edge, but the edge has become "thick" due to the thresholding. However, since we know the edge occurs at the peak, we can localize it by computing the maximal value of magnitude.

In practice, edge detection very often is performed by convolving the image with an appropriate kernel in the spatial domain [74], because it is computationally less expensive and often yields better results. However, for the purposes of opacity function generation, the spatial location or orientation of the edges is not relevant. The opacity function applied to the volume will be applied everywhere the same way: it is a single function that assigns opacity with no respect to position. Therefore, the goal is to locate boundaries not in the *spatial* domain, but in the *data value* domain.

Another difference between opacity function generation and the task of edge detection is that volume visualization works with a limited class of data. In order for a dataset to be visualized by direct volume rendering, there should be some correlation between the data value and the structure of interest, otherwise no transfer function will be successful. On the other hand, this is not a necessary restriction for edge detection, since the inevitable variations in illumination mean that individual objects (and their edges) take on a wide range of values. While these kinds of intensity fluctuations are naturally handled in edge detection, direct volume rendering of a volume dataset with similar data value variations would be very difficult.

Different edge detection kernels calculate either the 1st or the 2nd derivative of a two-dimensional image. The kernel producing the maximum response at a pixel location determines the edge magnitude and orientation. Different sets of kernels might be used: examples include Prewitt, Sobel, Kirsch and Robinson kernels [77].

The operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image.

4.1.2. Sobel Operator

An efficient edge detecting operator should have the following properties:

- It should be isotropic.
- It should have good localization
- It should have good signal-to-noise characteristics
- It should have accurate determination of edge orientation

The Sobel [12] gradient operator almost fully satisfies all the requirements, and is simple to implement. The Sobel operator is a little slower to compute than the Roberts [12] Cross operator, but its larger convolution kernel smoothes the input image to a greater extent and so makes the operator less sensitive to noise. The Sobel operator also generally produces considerably higher output values for similar edges.

As with the Roberts Cross operator, output values from the operator can easily overflow the maximum allowed pixel value for image types that only support small integer pixel values (*e.g.* 8-bit integer images). When this happens the standard practice is to simply set overflowing output pixels to the maximum allowed value. The problem can be avoided by using an image type that supports pixel values with a larger range. Natural edges in images often lead to lines in the output image that are several pixels wide due to the smoothing effect of the Sobel operator.

The Sobel algorithm convolves the incoming image data set, an example of the typical window is given in Figure 34, with four 3x3 masks, shown in Figure 35; that is, Sobel operators, weighted to measure the differences in intensity along the horizontal(H), vertical(V) and left (DL) and right(DR) diagonal directions.

Consider an arrangement of pixels about an arbitrary pixel [i,j] as shown in Figure 34. The window consists of nine elements, the current pixel [i,j] and its eight nearest neighbors, represented by a0….a7. Figure 36 shows the way this window is updated in one clock cycle.

$a0$	$a1$	$a2$
$a7$	$[i,j]$	$a3$
$a6$	$a5$	$a4$

Figure 34 Typical arrangement of a window of pixels for an image

$$H = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad V = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$DL = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \qquad DR = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

Figure 35 Sobel convolution kernels

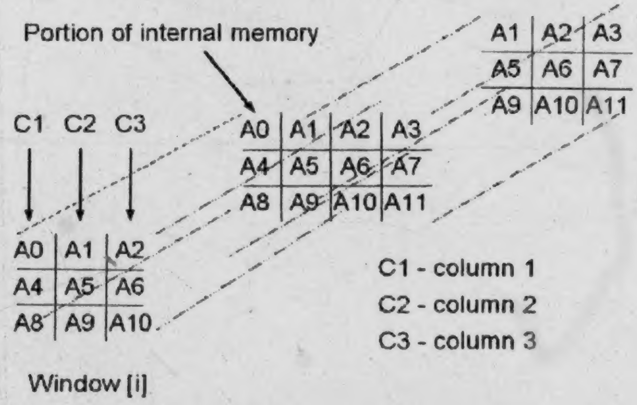


Figure 36 Window updating

The direction of the edge, also called the phase of edge, is represented by a 3-bit vector as shown in Figure 37 and in Table 4. The edge direction is defined to be the direction associated with the maximum intensity variation.

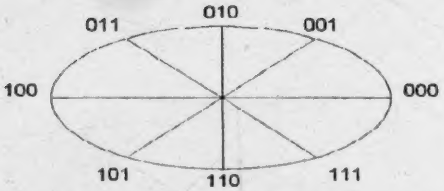


Figure 37 Direction representation

Table 4 Codes for edge detection

Direction (Phase)	Code
Positive Horizontal	000
Negative Horizontal	100
Positive Vertical	010
Negative Vertical	110
Positive Right diagonal	001
Negative Right diagonal	101
Positive Left diagonal	011
Negative Left diagonal	111

Figure 38 illustrates the results of application of the corresponding Sobel convolution kernels shown in Figure 35 for the image presented in Figure 12.

The next section shows mathematical aspects of the Sobel operator and how the gradient magnitude is derived. If the greatest magnitude exceeds a specified threshold, the pixel in the center of the window is declared to be part of an edge.

4.1.2.1. Mathematical Derivations

These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation. These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient.

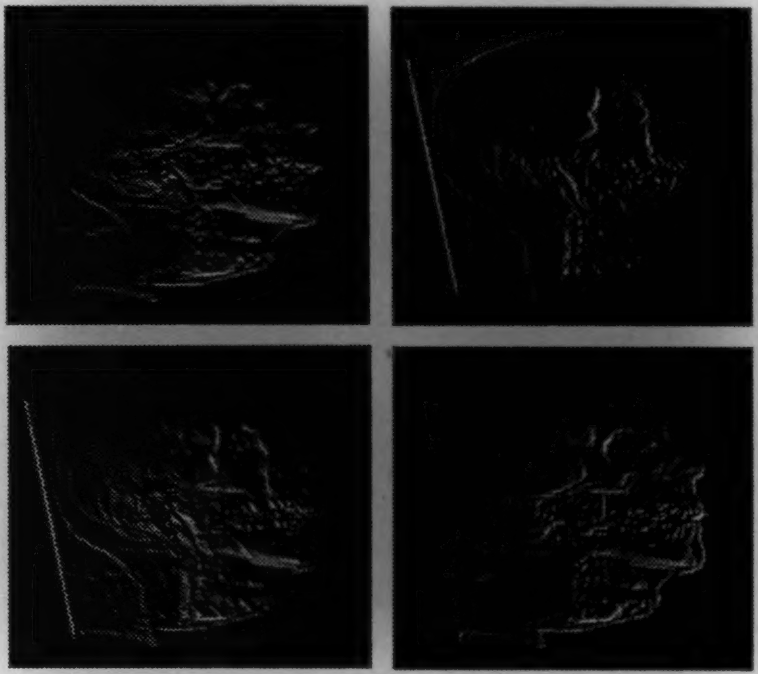


Figure 38 The results of corresponding Sobel convolution kernels applications

Horizontal Sobel convolution kernel

$$E(H) = (a0 + 2a1 + a2) - (a4 + 2a5 + a6)$$ **Equation 22**

Vertical Sobel convolution kernel

$$E(V) = (a2 + 2a3 + a4) - (a0 + 2a7 + a6)$$ **Equation 23**

Left diagonal direction Sobel convolution kernel

$$E(DL) = (a1 + 2a2 + a3) - (a7 + 2a6 + a5)$$ **Equation 24**

Right diagonal direction Sobel convolution kernel

$$E(DR) = (a1 + 2a0 + a7) - (a3 + 2a4 + a5)$$ **Equation 25**

In this case, orientation 0 is taken to mean that the direction of maximum contrast from black to white runs from left to right on the image, and other angles are measured anti-clockwise from this.

The gradient magnitude is given by: **Equation 26**

$$Magnitude = Max\left[|E(H)|,|E(V)|,|E(DR)|,|E(DL)|\right]+\frac{1}{K}\lfloor|E(\perp)|\rfloor$$

where $\lfloor|E(\perp)|\rfloor$ is the absolute value of the intensity variation in the direction, perpendicular to the maximum intensity variation.

What is the best value of K [37]?

The gradient direction is assigned from a template of eight angle directions based on the selected maximum measure and its corresponding sign.

$$\theta = \arctan\left[\frac{E(V)}{E(H)}\right]$$ **Equation 27**

In most cases, the determination of the value of K is application dependent. Although if the value, that minimizes the magnitude error will determine the choice of K, it might be calculated using the error analysis or Euclidian norm approximations.

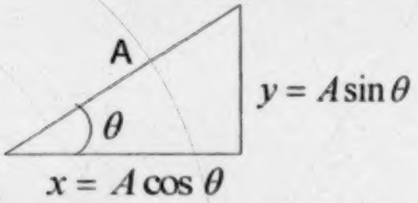


Figure 39 Euclidian normal approximation

Consider Figure 39, the magnitude A is given by

$$A = \left[x^2 + y^2\right]^{\frac{1}{2}}$$ **Equation 28**

$$A = \frac{y}{\sin\theta} = \frac{x}{\cos\theta}$$ **Equation 29**

For $0 < \theta < \frac{\pi}{4}$, we have x > y, and A can be approximated by

$$\hat{A} = x + Ky$$ **Equation 30**

or
$$\hat{A} = \left[\frac{x}{y} + K\right] \bullet y$$
Equation 31

or
$$\hat{A}(\theta, K) = y\left[\frac{1}{\tan\theta} + K\right]$$
Equation 32

where 0< K<1. The error percentage in this case can be calculated by

$$E[\theta, K] = 100\left[\frac{\hat{A}(\theta, K) - A(\theta)}{A(\theta)}\right]$$
Equation 33

Combining the Equations 29 – 33 we can get the expression given below:

$$E[\theta, K] = 100(\cos\theta + K\sin\theta - 1)$$
Equation 34

where $0 < \theta < \frac{\pi}{4}$ and 0< K < 1.

The graph in Figure 40 illustrates the error percentage for different values of K and θ, and indicates that a minimum error for both 0-22.5° range (i.e., four Sobel operators) and 0-45° range (i.e., two Sobel operators) is obtained for K=1/4. Therefore, K will be chosen to be 1/4.

An example below illustrates the magnitude and phase computations.

Assuming the next 3x3 window:

12	01	08
05	09	03
40	03	10

After application of the four Sobel masks to this image to compute the intensity variations along the four directions, the following results are obtained:

E(H) = (-34); E(V) = (-38); E(DR) = (+4); E(DL) = (-68)

Therefore, the maximum value of |-68| corresponds to the left diagonal direction. Since the value is negative, the direction of maximum intensity variation is 111. Since a right diagonal is perpendicular to a left diagonal, then $E(\perp) = E(DR) = 4$.

According to Equation 26, the gradient magnitude is:

$$Magnitude = Max\left[\,|-34|,|-38|,|4|,|-68|\,\right]+\frac{1}{4}\lfloor|4|\rfloor = 68+1=69$$

If the threshold is <69, the pixel in the center of an example window is declared to be a part of a left diagonal edge with direction 111.

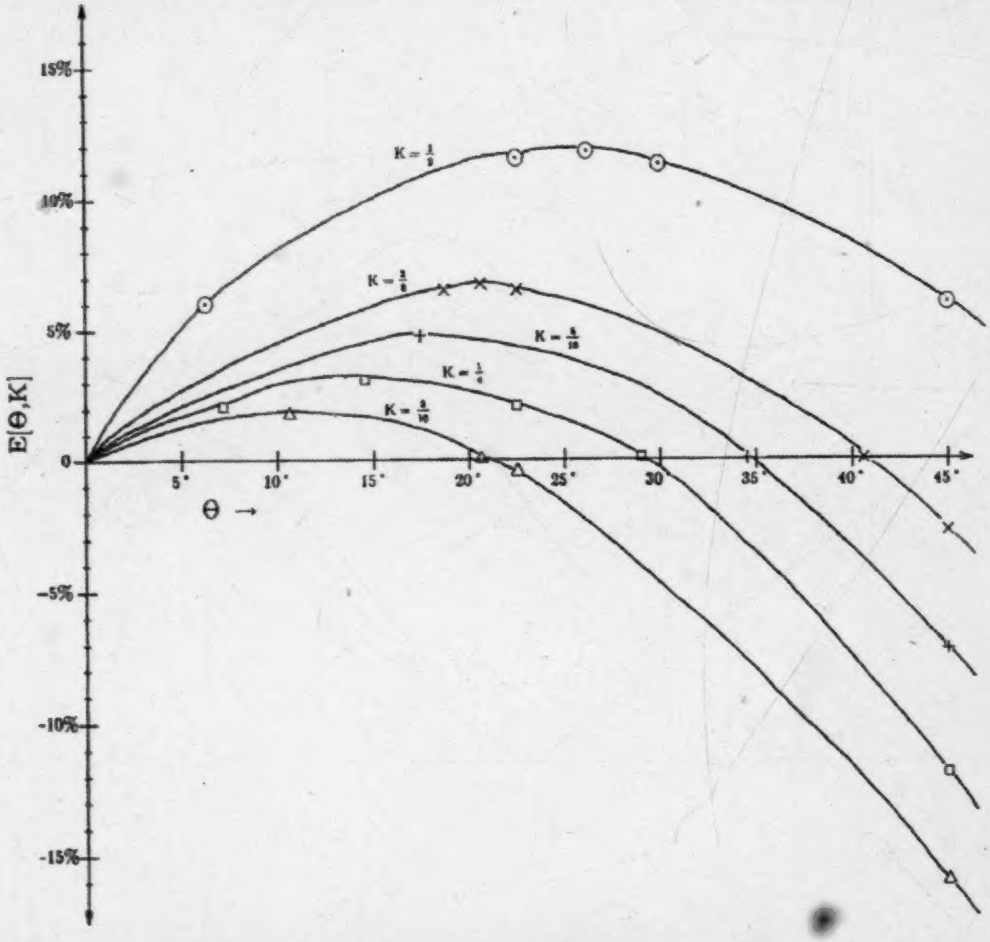


Figure 40 Error for approximating the magnitude function

After edge detection each image is assigned to the foreground value (255), while non-edge pixels are assigned to the background value (0). In this example, the center pixel in the window was declared to be the part of an edge and its value set to be the foreground value (255). As shown in Figure 41 edges appear as white lines on the black background.

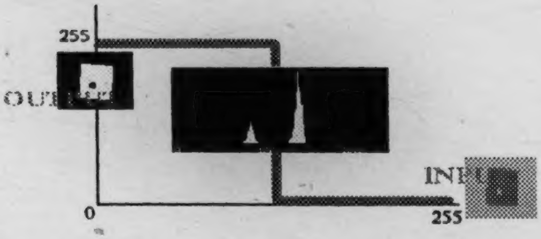


Figure 41 Edge detection boundaries

4.1.2.2. Previous Work

The majority of Sobel operator edge detection functions for image processing and volume rendering purpose have software implementations, and just very few of them have hardware implementations.

Kanopoulos in [37] described the chip targeted for image edge detection filter using the Sobel operator. Figure 42 shows the functional block diagram of that magnitude and direction processor. It has initial computational delay of 10 clock cycles, after which outputs are produced every clock cycle (clocked with 10 –MGz clock), 5 pipeline stages, and equivalent throughput of 200x(10^6) additions/s. The design was compiled in 2um CMOS technology with double metal interconnections using GENESIL compiling system.

The main drawback of this design is in its quite irregular structure. The Sobel operator processor in this work avoided this problem by the optimizations made at the mathematical model level.

4.1.3. Sobel Operator Implementation

The Sobel edge detection block detects edges in the SWF volume rendering system using the Sobel operator. The volume voxels (image pixels) are supplied in the raster scan order (i.e. rows are scanned left to right and top to bottom). The edge detector serially outputs data, which contain the edge information: magnitude and direction. The block should be synchronized with a system clock (SCLK), and designed in such a way that images of various sizes can be processed.

The edge detection algorithm proceeds with the three following steps:

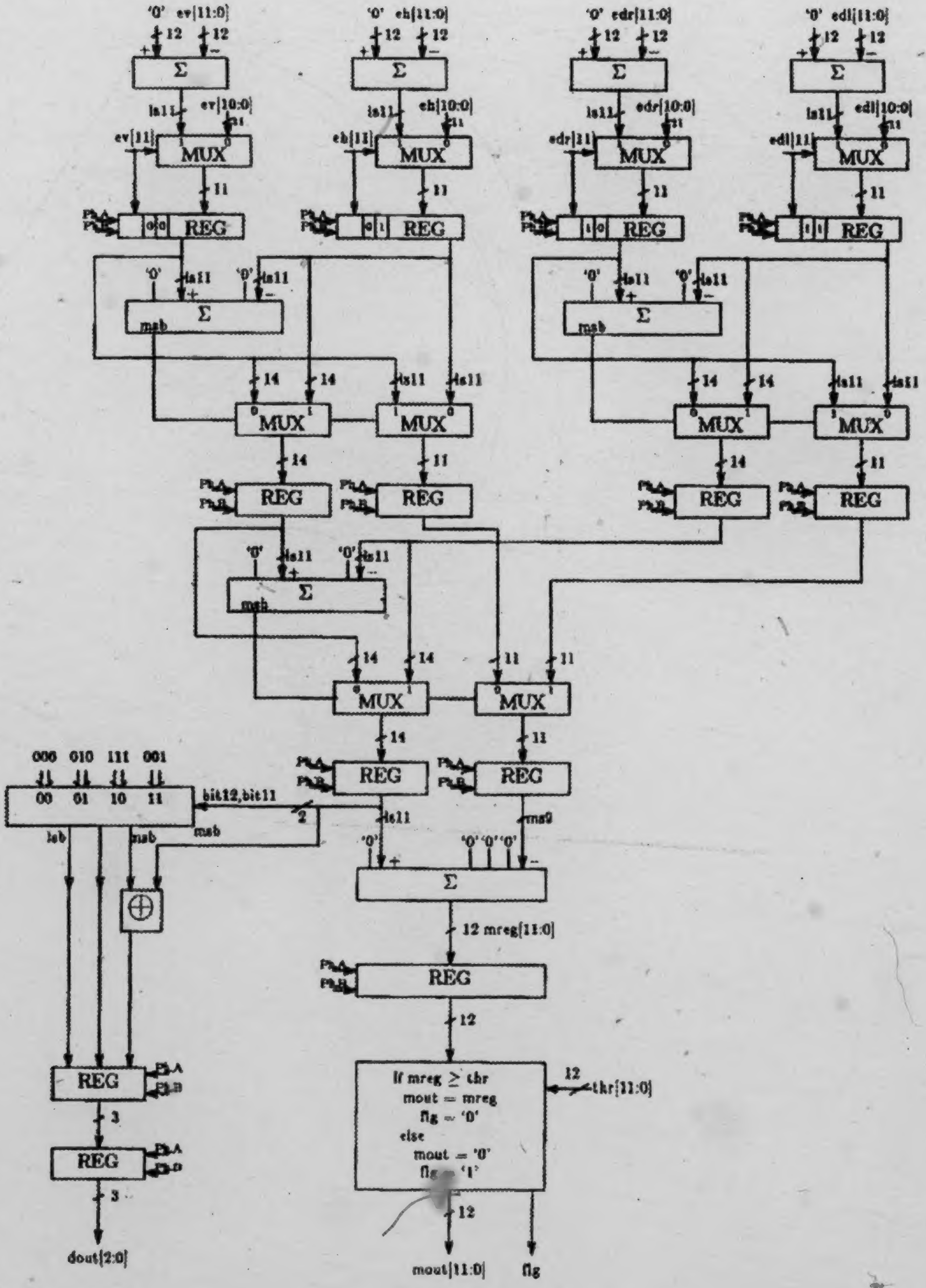


Figure 42 Functional block diagram of the magnitude and direction processor

Step1. During each clock cycle update the internal memory data structure of the corresponding context RAM. Pixels arrive in the raster scan order with one pixel arriving at the input port INPUT every clock period. This pixel value must be stored in the 3-row internal memory array. When the entire window has been constructed with the arrived pixels, the filter window buffer must be updated according to Figure 36.

Step 2. Compute the four Sobel filter functions (H, V, DL and DR) on the current filter window using the pixels in the updated window buffer.

Step 3. Compute maximum absolute value for the four filter outputs, choose the absolute value of the perpendicular, and compute the magnitude according to Equation 26.

. If the magnitude is greater than the specified threshold, the central pixel in the current window is declared to be an edge pixel. Therefore, the appropriate pixel value is assigned and direction is chosen according to Figure 37.

4.1.3.1. Sobel Edge Detection Processor Architecture

The top level Sobel operator processor structure is shown in Figure 43.

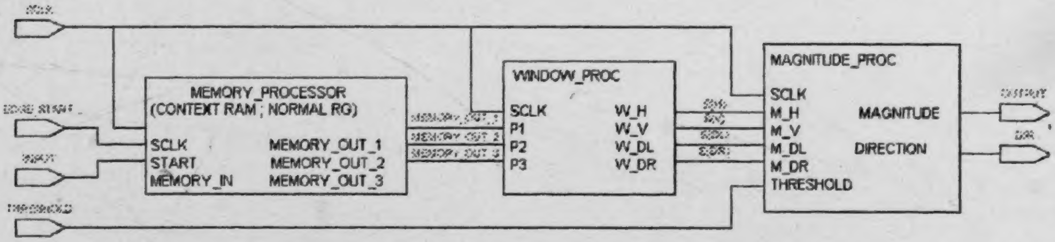


Figure 43 Top level decomposition of the Sobel edge detection system

Figure 44 shows the hierarchical Sobel operator processor structure built by structural components and described by corresponding files.

Memory processor.

Pixels arriving at the port MEMORY_IN during each SCLK cycle are stored into the appropriate location of the internal context memory buffer. Through ports MEMORY_OUT_1, MEMORY_OUT_2, and MEMORY_OUT_3 the appropriate 3-pixel column read from the Context RAM, and sent through the corresponding normal registers to the Window processor for the current window updating. The system START signal initializes the memory context snooping when a new slice is to be processed.

Window processor.

The Window processor contains a 3x3 window buffer. Each clock cycle delivers a new column of data from its inputs: P1, P2 and P3, and updates the window buffer as shown in Figure 36. Then, it computes four filter functions and outputs the four intensity gradients at ports W_H (horizontal intensity gradient), W_V (vertical intensity gradient), W_DL (left diagonal intensity gradient), and W_DR (right diagonal intensity gradient).

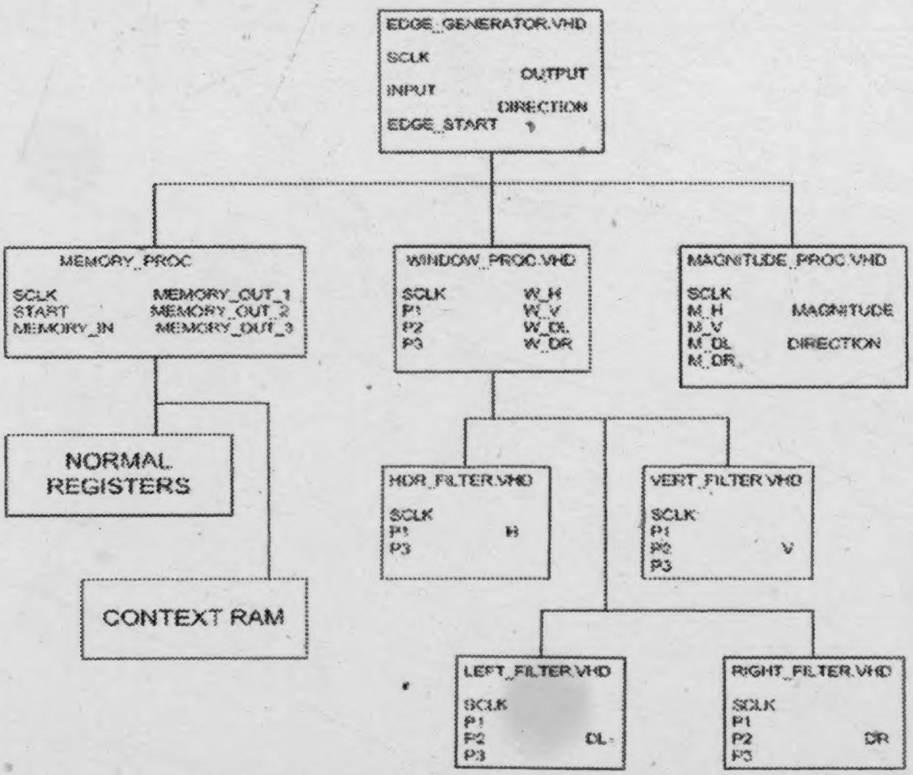


Figure 44 Hierarchical decomposition of the Sobel operator edge detecting system

Magnitude processor.

The Magnitude processor reads intensity gradients passed to its input ports from the output ports of Window processor, respectively, W_H, W_V, W_DL and W_DR to M_H, M_V, M_DL and M_DR, and computes the magnitude gradient according to Equation 26, and the direction of the maximum gradient according to Figure 37.

If the magnitude exceeds the threshold (THRESHOLD), the center pixel of the window is declared as an edge pixel, and MAG_OUT is set to the foreground value, and DIR is set to the direction of the maximal gradient, otherwise, the MAG_OUT is set equal to the background value, and DIR is arbitrary.

As shown, the Memory processor is structurally built from the Context RAM, which is written and read through normal registers. The Window processor decomposed into four filter components: horizontal filter (HOR_FILTER), vertical filter (VER_FILTER), left diagonal filter (LEFT_FILTER) and right diagonal filter (RIGHT_FILTER).

4.1.3.1.1. Design of the Memory Processor

The memory array represents the internal voxel memory. It is an array with three rows. The number of columns is the number of voxels in each row of the image (signal NUM_COLS) as shown in Figure 45. A particular window is shown in Figure 46.

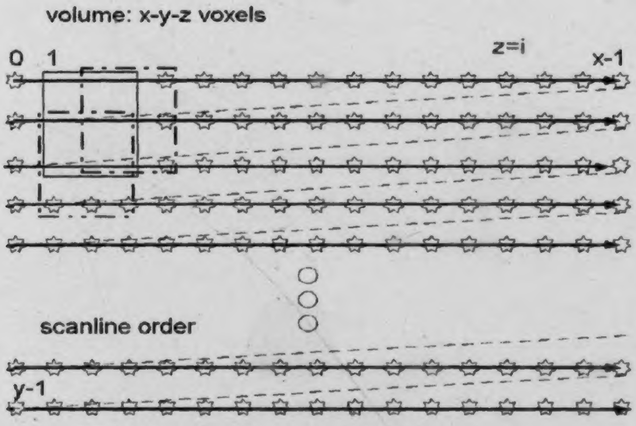


Figure 45 Memory array

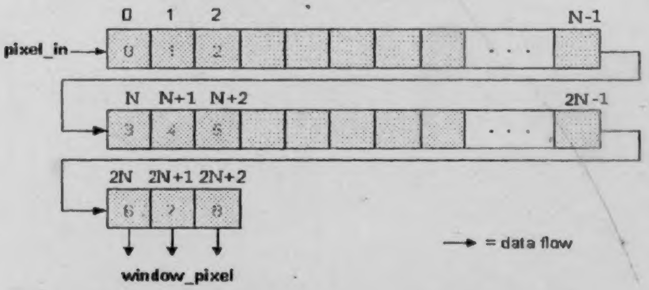


Figure 46 Volume slice (image) window

The memory processor stores three rows x three columns voxels of information in its internal memory. Memory is built in FFs with synchronous reset and outputs the first column, renewed after window updating as shown in Figure 36. The memory controller model (RAM chip file) for RD and WR is given in Appendix 2.

The memory is used in a round robin fashion: according to window update (Figure 36). The next column is loaded from the dataset stored in the Context RAM. When the row counter has achieved its maximum value, the window is updated in a scan line order, as shown in Figure 45.

4.1.3.1.2. Design of the Window Processor

The filter component can be calculated according to the general equation given below:

$$E(F) = (x + 2y + z) - (k + 2l + m) \quad \textbf{Equation 35}$$

The most common expression $(x + 2y + z)$ forms the basic Sobel operator cell shown in Figure 47. The filter magnitude can be computed according to the schematic given in Figure 48.

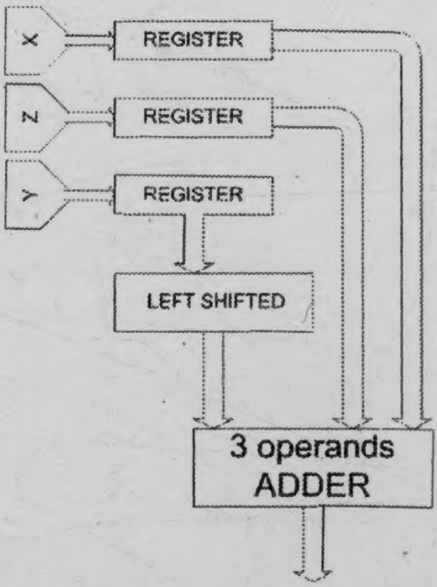


Figure 47 Basic Sobel operator cell

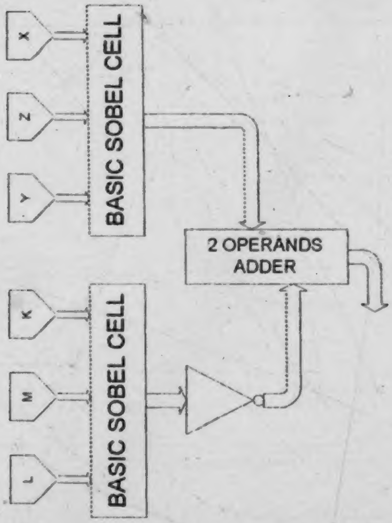


Figure 48 Filter magnitude calculator block diagram

Consideration must be given to the following: frequently, the gray level of a pixel is assigned to be a value in the range of 0 to 255, with 0 corresponding to black, 255 corresponding to white, and shades of gray distributed over the middle values. Thus, eight bits are required to represent the gray level, which is always a positive integer. Since filters are calculated using Equations 22-25, two more bits are required to be able to represent the partial summation magnitude. The subtraction operation can create a negative number; therefore, an 11-bit representation is needed for the output. In order to use a standard bus, 12 bit two's compliment vectors are used for the filters' output representation.

Since left shift operation equals multiplication by 2, a three-operands adder sums X, Z and shifted left Y operands. The output of the basic cell is the partial magnitude. Filter magnitude can be calculated by subtraction of two partial magnitudes, where the negative partial magnitude is simply added as the two's compliment, and carry in of the two operands adder is set to "1".

The Window processor consists of the following components:

The horizontal filter component computes the horizontal filter operation according to Equation 22. The current window top row pixels arrive at the input port P1 and the bottom row – at the input port P3. The middle row pixels are not needed according to the horizontal convolution kernel illustrated in Figure 35. One pixel arrives at each port every clock cycle. This component keeps its own internal copy of rows 1 and 3 of the buffer window.

The vertical filter component computes the vertical filter operation according to Equation 23. The current window top row pixels arrive at the input ports P1, the middle - at the port P2 and the bottom row of pixels arrives at the port P3. One pixel arrives at each input port every clock cycle. This component keeps its own internal copy of the all three rows of the buffer window.

The left diagonal filter component computes the left diagonal filter operation according to Equation 24. The current window top row pixels arrive at the input ports P1, the middle-at the port P2 and the bottom row of pixels arrives at the port P3. One pixel arrives at each input port every clock cycle. This component keeps its own internal copy of the all three rows of the buffer window.

The right diagonal filter component computes the right diagonal filter operation according to Equation 25. The current window top row pixels arrive at the input ports P1, the middle-at the port P2 and the bottom row of pixels arrives at the port P3. One pixel arrives at each input port every clock cycle. This component keeps its own internal copy of the all three rows of the buffer window.

4.1.3.1.3. Design of the Magnitude Processor

The magnitude calculation algorithm is as the following:

Step 1. By comparing the pairs of kernel values define the maximal intensity variation. Comparison is done in parallel on perpendicular kernels pairs: E(H) and E(V), E(DL) and E(DR). That fact that the gradients in pair are perpendicular to each other is used in the Step 3. The comparison starts from the MSB until the bigger value in pair is detected. It is further compared to the bigger value from the other pair. The result is the maximum intensity gradient. In such a way we reduce the number of operands to be compared from (3/2 n) combinations to (n-1).

Step 2. Three-bit left shift.

Step 3. Define the intensity variation in the direction, perpendicular to the maximum intensity variation.

Since the gradients are firstly compared to its perpendicular kernels, and then the final maximum gradient value is detected, so the intensity variation in the direction perpendicular to the maximum intensity variation is simply picked from the first comparison stage as the other corresponding gradient.

Step 4. Add the absolute value of the intensity variation in the direction, perpendicular to the maximum intensity variation.

Step 5. Three-bit right shift.

Step 6. Compare to the threshold value and define the direction.

The comparator compares the final value with an input threshold value, and defines the pixel status: edge pixel or not, and its direction.

Figure 49 shows the structural diagram of the Magnitude processor.

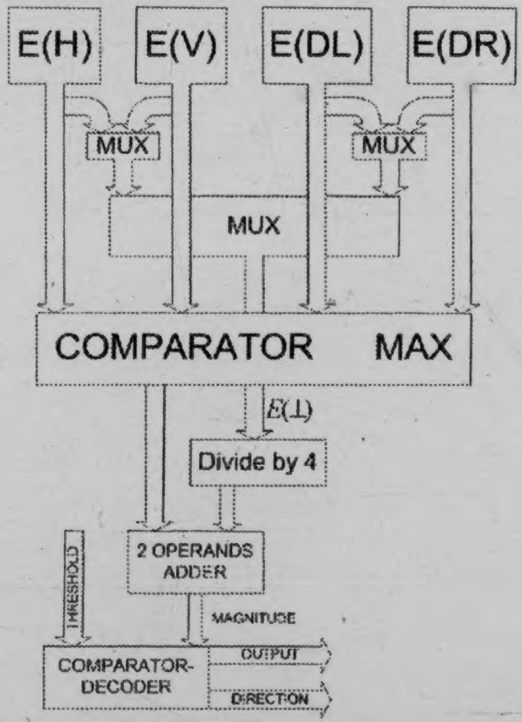


Figure 49 Sobel operator magnitude and direction processor block diagram

The Sobel edge detection system has a VHDL implementation. It should be pointed out that single holes might be either filled or ignored; multiple holes and multiple corner holes remain unfilled.

The entity is given below:

```
entity edge_detector is
  generic(threshold: filter_out;  -- a threshold depend on which we define the
                                  -- edge pexels
          num_rows:natural;        -- the number of rows of the image file
          num_cols: natural);      -- the number of columns of the image file
  port(clock: in Std_ulogic;       -- the system CLOCK
     edge_start : in std_logic:='0'; -- the START signal for the image processing
     input: in pixel;              -- input pixel
     output: out pixel;          -- output pixel
     dir  : out direction);            -- output direction
end edge_detector;
```

Structural description of the Sobel edge detector is shown below:

```
architecture STRUCTURE of EDGE_DETECTOR is
  -- memory processor component declaration --
  component MEMORY_PROCESSOR1
    generic (NUM_ROWS, NUM_COLS: NATURAL;
           MEM_OUT_DELAY:TIME)
    port(CLOCK: in STD_LOGIC:='0';   -- system CLOCK
        START: in STD_LOGIC:='0';   -- start signal
        MEM_IN: in PIXEL:=0;        -- input pixel
        MEM_OUT1, MEM_OUT2, MEM_OUT3: inout PIXEL:=0);
  end component;

-- window processor component declaration --
  component WINDOW_PROCESSOR1
    generic(HORIZ_DELAY, VERT_DELAY, LEFT_DIAG_DELAY,
           RIGHT_DIAG_DELAY, WAIT_TIME:TIME)
  port(CLOCK: in STD_LOGIC:='0'; -- system CLOCK
     P1,P2,P3: in PIXEL:=0;      -- input pixels
     W_H: inout FILTER_OUT:=0; -- horizontal filter
     W_V: inout FILTER_OUT:=0; -- vertical filter
     W_DL: inout FILTER_OUT:=0; -- left diagonal filter
     W_DR: inout FILTER_OUT:=0);-- right diagonal filter
 end component;

-- magnitude processor component declaration --
 component MAG_PROCESSOR1
   generic(MAG_DELAY: TIME); -- magnitude processor delay
   port(CLOCK: in STD_LOGIC:='0'; -- system CLOCK
      M_H,M_V,M_DL,M_DR: in FILTER_OUT:=0; --filter
       -- values from the window processor
      THRESHOLD: in FILTER_OUT:=0; -- threshold value
      MAG_OUT: inout PIXEL:= 0; -- edge pixel value
      DIR: inout DIRECTION :="000"); -- edge direction
 end component;

-- intermediate signals between the components --
 signal E_H,E_V,E_DL,E_DR: FILTER_OUT:=0;
 signal MEM_OUT1, MEM_OUT2, MEM_OUT3: PIXEL:=0;
```

The following basic package components are described in the file rtl_pack.vhd:

half_adder, full_adder, sum12_pp, full_substractor, sum12_pm, reg1,reg3, reg8, reg12, mux1_by2, Mux12_by_2, abs12,reg_angle, divider and compare GE.

4.1.4. Sobel Operator Processor Verification

RTL code was compiled using FPGA Advantage tools, and exhaustively tested.

The hierarchical Sobel edge detection system has been tested in a bottom-up fashion, in the same manner as it was designed: down –up. Individual components as well as the aforementioned package components were tested using separate testbenches to verify the behavioral performance of the each leaf module.

Below in Figure 50 is a hand drawn image according to the computed results produced from the recognizable outline of a real image.

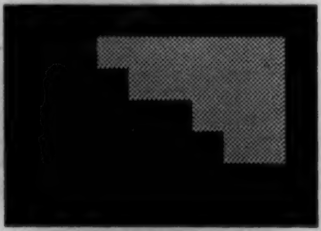


Figure 50 The Image

To compare, the image in Figure 51 is generated using MATLAB after applying of the Sobel masks. These images are zoomed in 16 times.

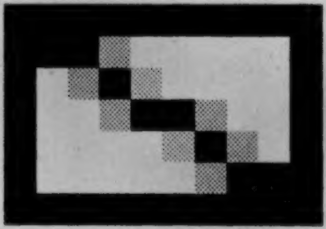


Figure 51 Sobel edges of the Image

That image, 10x7 pixels with bi-linear re-sampling, was chosen to simulate and to verify functionality. The memory processor was emulated to test the other components. More than 140 testbenches has been set up for the full Sobel edge detector verification. The verification results are as the following:

- H, V, DL and DR edges are detected with correct direction
- Corner pixels are detected correctly
- Single and multiple holes are processed correctly
- Edges in the tested image in are identified.

The Table 5 below shows the testing results obtained after the number of testbenches set up for the Image edge detection:

Table 5 The Image edges computed by Sobel edge detector

	Width pixels									
Height pixels	x LT	xT	xT	xT	xT	xT	xT	xT	xT	x RT
	xL	x MH	xMH							xR
	xL			xMV						xR
	xL				xMH	xMH				xR
	xL						xMV			xR
	xL							xMH	xMH	xR
	x LB	xB	xB	xB	xB	xB	xB	xB	xB	x RB

Where: L-left, R-right, T-top, B-bottom, M-middle, V-vertical, H-horizontal edges.

Table 5, as seen, fully corresponds to the Figure 51 obtained after Sobel masks application using MATLAB. In this way we have verified the Sobel operator processor including boundary cases.

The VHDL code is easily synthesized using Synopsis. The filter code was synthesized in CMOS 0.18 um technology to prove its workability. Standard AVANT 0.18 cells were used. One pipelining stage was asserted after three operands adder, and clocked by the SCLK. According to the table below the maximal propagation delay is 2.42 ns; considering this as the initial delay, the next output is produced every clock cycle.

Table 6 Filter circuit performance

Clock period	2.5 ns
Maximum circuit delay	2.42 ns
Power consumption of the actual circuit	31.28 uW

Some problems occurred with synthesis tool, since some VHDL constructs were not supported, such as generic DELAY and TIME declaration, and AFTER clauses as well.

4.2. Matrix Multiplication Processor

4.2.1. Application Aspects

Fast matrix multiplication algorithms such as Winigrad's or Strassen's methods are targeted to reduce the number of computations. However, hardware implementations using those approaches are quite complicated, and might be accepted only for relatively big size matrices.

The proposed matrix multiplication processor, mainly developed for the volume rendering applications, is based on inner products evaluations. A 1x4 matrix requires computation of three 3D inner products: *i, j* and *k*.

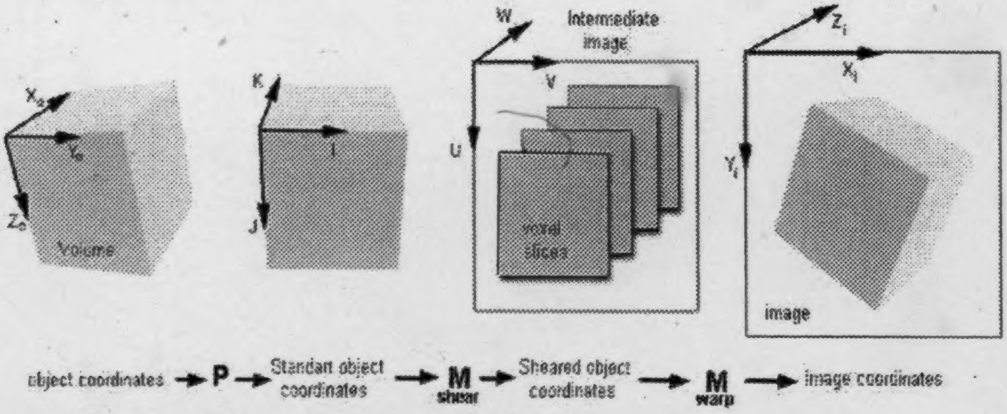


Figure 52 Shear-Warp Factorization coordinate transformation

As shown in Figure 52, the shearing matrix *Mshear* transforms the image into sheared coordinates with similar computations of two 4 x 4 matrices, and, therefore, requires to estimate 16 inner products. In turn, the warping matrix *Mwarp* converting the sheared image into final image coordinates uses 9 inner products. These numbers show the necessity to engage an accelerating technique(s). Employing pipelining and parallelism can, obviously, help with this problem.

The matrix multiplication processor given in Figure 53 should be developed to compute 4 -elements matrixes.

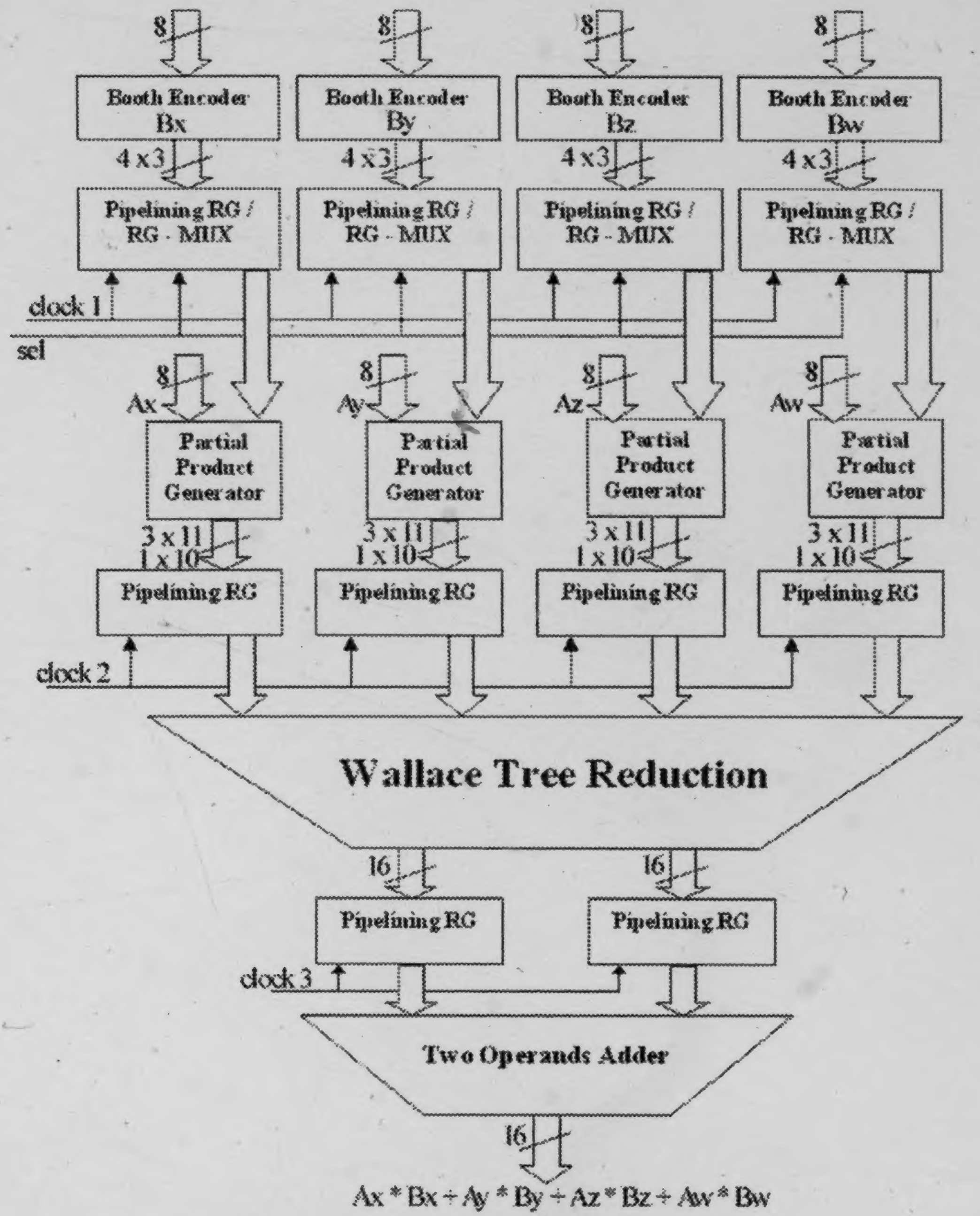


Figure 53 Matrix multiplication processor structure

The equation below clearly illustrates that the operands in dot products are repeated correspondingly to the number of columns and rows. Thus, pipelining is an efficient technique to reduce the latency time.

$$\begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} * \begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{bmatrix} =$$

$$= \begin{bmatrix} a11b11+a12b21+a13b31+a14b41 & \ldots \ \ldots & a11b14+a12b24+a13b34+a14b44 \\ a21b11+a22b21+a23b31+a24b41 & \ldots. \ \ldots & a21b14+a22b24+a23b34+a24b44 \\ a31b11+a32b21+a33b31+a34b41 & \ldots \ \ldots & a31b14+a32b24+a33b34+a34b44 \\ a41b11+a42b21+a43b31+a44b41 & \ldots \ \ldots & a41b14+a42b24+a43b34+a44b44 \end{bmatrix}$$

Equation 36

Equations in Chapter 1 use 4 x 4 transformation matrix. However, practically the only a processor to compute 3 x 3 matrix is needed since in the shear-warp transformation matrices the last column or row has "0s", or give "0s" after multiplications. Moreover, a matrix sized 3 x 3 is used for the Sobel's Operator evaluation, in the Shading processor, and in the Warping units (8-bit operand representation).

Therefore, the matrix multiplication processor to compute 3 x 3 8-bit representation matrices is required.

4.2.2. MMP Design and Verification

To achieve the necessary matrix processing speed to provide an interactive image rendering a fast inner processor is required. The following chapter presents an 8-bit representation signed 3D operands Inner Product Processor (IPP).

The entire architecture of the Matrix Multiplication Processor consists of two parts: an Inner Product Processor and a Controller. The IPP is the main component of the Matrix Multiplication processor (MMP), and the rest of the MMP "delivers" data from the corresponding Context RAM to the IPP and normal registers, defines the terms and number of terms to be added (from 0 to 3), and a working mode. Data retrieved from the volume window are stored in the Context RAM, and transformation matrixes are stored in look up tables, accessed, again, through the normal registers. Figure 54 shows the MMP controller diagram.

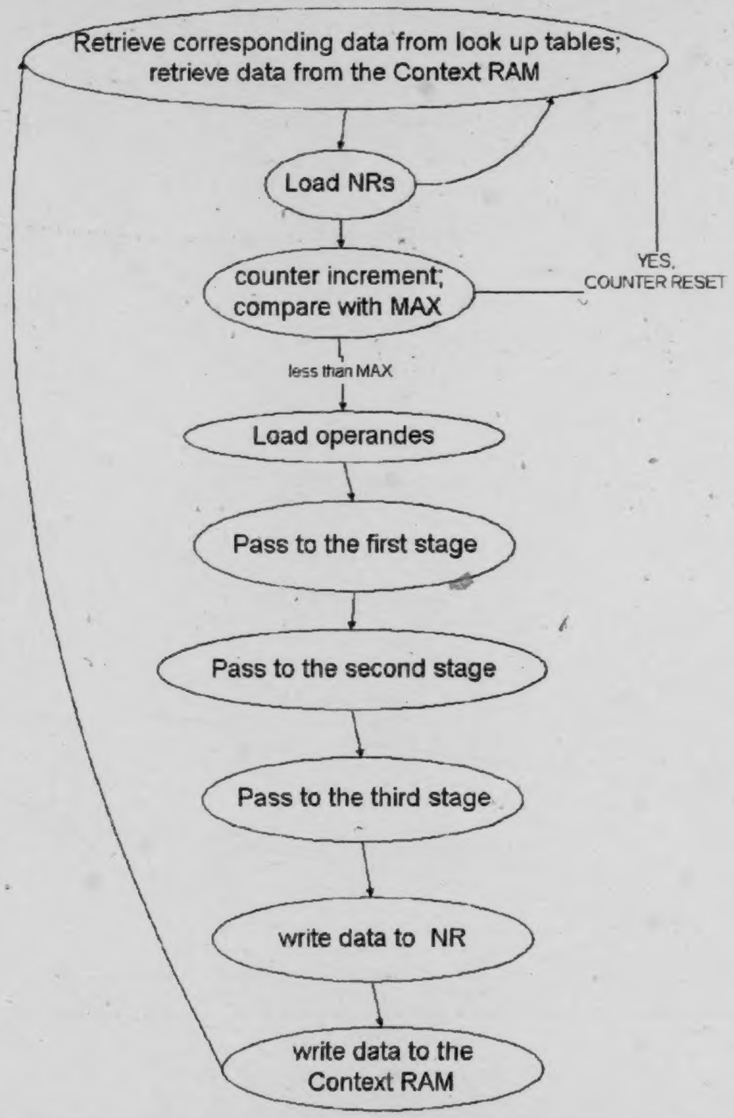


Figure 54 MMP controller diagram

Data are obtained from the Context RAM and look up tables, and delivered through the normal registers to the IPP. A counter regulates the matrix size from 1 to 16. Pipelining aspects are discussed below. Three pipeline stages are used. Tasks scheduling (an example for 9 scalar products) is shown in Table 7.

The MMP controller is "embedded" into the VRS controller in terms of context data snooping and normal registers loading.

The Composer is described in the following section. The IPP itself is implemented in silicon, and described in detail in the following chapter.

Table 7 Pipelining tasks scheduling

Clock Cycles	Stages		
	S1	S2	S3
1	I1-1		
2	I2-1	I1-2	
3	I3-1	I2-2	I1-3
4	I4-1	I3-2	I2-3
5	I5-1	I4-2	I3-3
6	I6-1	I5-2	I4-3
7	I7-1	I6-2	I5-3
8	I8-1	I7-2	I6-3
9	I9-1	I8-2	I7-3
10		I9-2	I8-3
11			I9-3

This design can be adjusted for general matrix multiplication. It can be easily extended to process any operands of any bit representation inner product, or, in other words, to multiply any size matrixes. Figure 55 shows the architecture of 16-bit representation Inner Product Processor with Partially Merged Technique [13] utilization. Employment of the Partially Merged technique for the Wallace tree reduction will increase the number of 4-2 compressors required, thus, increasing power consumption; however, it will speed up the computation of a lager numbers of partial products as well as terms.

The MMP has two implementations: VHDL with pipelined architecture, shown in Figure 53, tested in PeakVHDL and verified. The other VLSI implementation does not include pipelining within the IPP.

Retrieving data from the Context RAM through normal registers and writing back is provided by the Data Write/Read block. The mechanism is identical for the all VRS components. It takes two clock cycles, since we read/write simultaneously during the same cycle (dpr memory).

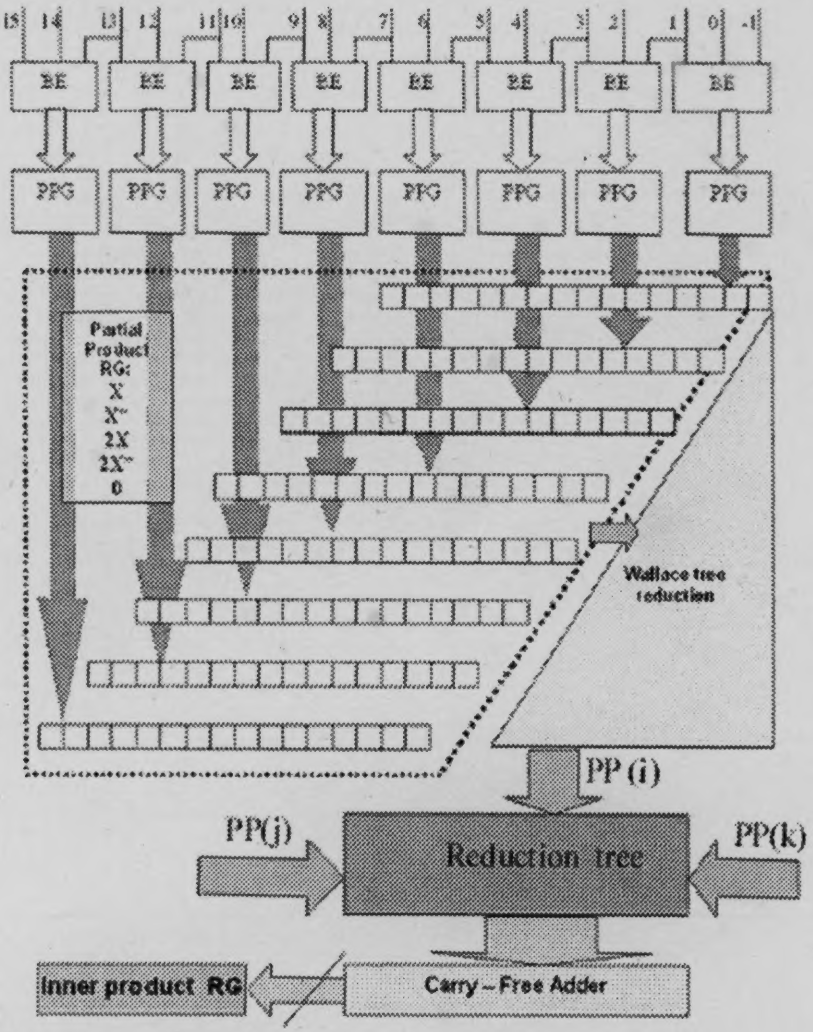


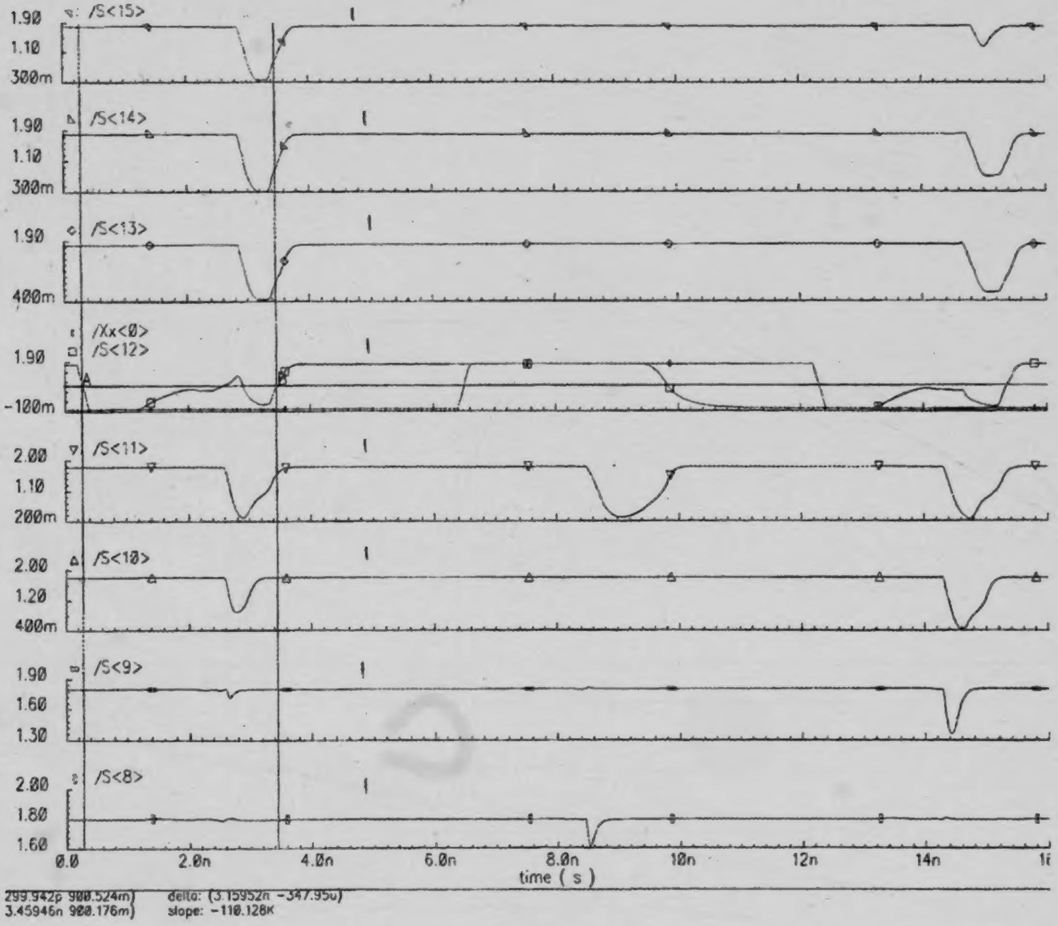
Figure 55 16-bit 3D operands IPP with Partially Merged technique

A four 8-bit terms Inner Product's Wallace tree with Fully Merged reduction requires 77 4-2 compressors; a three-term Inner Product requires 61 4-2 compressors.

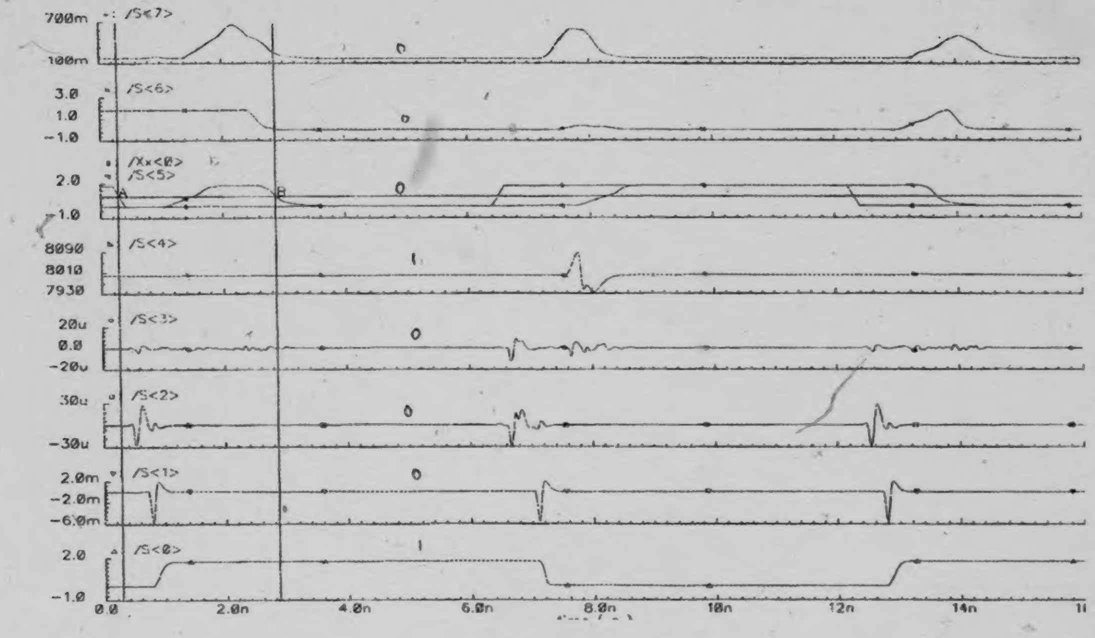
For the 8-bit operand representation the first stage of the Partially Merged technique utilizes 11 4-2 compressors, and the number of reduction stages depends on number of terms in the inner product and the way of columns' regularization. To compare; the Partially Merged reduction would increase the entire quantity of 4-2 compressors for three terms inner product from 61 to 73; for four terms - from 77 to 92, or 20 % more.

The non-pipelined MMP has been verified. The circuit achieves high-speed and low-power operation through enhanced multiplier structures at the architectural level and reduced swing CPL logic at the circuit level. The simulation showed the worst-case delay time of 3.16ns and an average power dissipation of 2.53mW at 1.8V and 200MHz.

Functional simulation results are shown below. Figure 56 1-2 gives the output results, and Figure 57 1-2 the inputs.

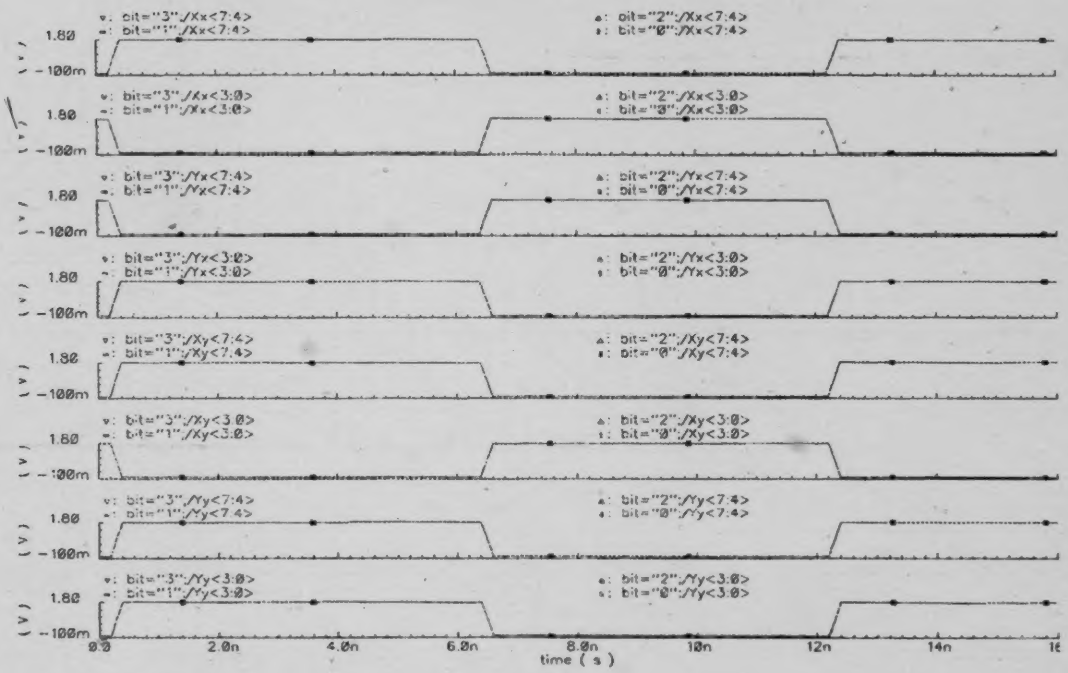


(1)

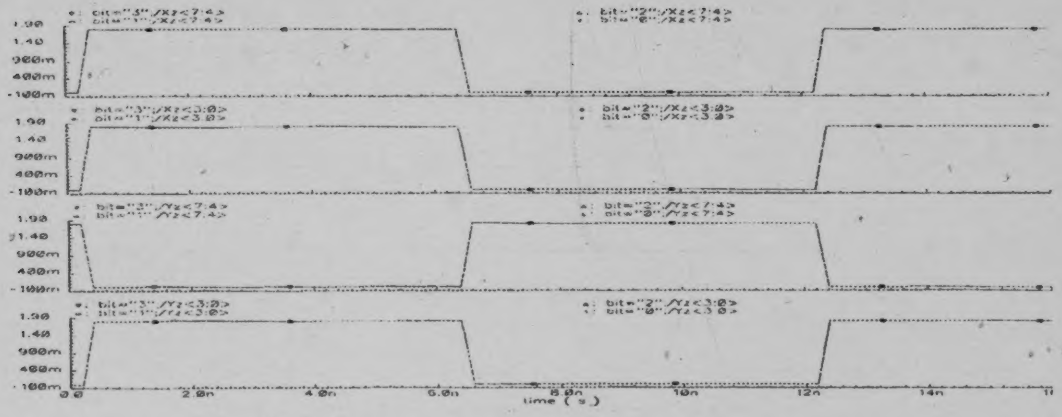


(2)

Figure 56 1-2 MPP functional verification: outputs



(1)



(2)

Figure 57 1-2 MMP functional simulation: inputs

The MMP is developed to compute 3 pairs of operands. However, dealing with 4 x 4 size volume rendering transformation matrix, as could be seen from the equations above, at least one operand is always equal "0". This pair of operands is truncated from the computation. This is done after operands are read from the corresponding Context RAM through a very simple mechanism when each operand is compared to "0", and that pair is excluded. In the case when in the each pair at least one operand is equal "0", the entire product calculation is skipped. This is considered as a boundary case.

The pipelined MMP as was mentioned has 3 arrays of pipelining FF. It was an attempt to insert only 2 arrays of FF: after the Booth Encoder, and after the Wallace Tree. The Booth Encoder has the biggest propagation delay- 1.47 ns. The first inner product is computed in 3. 36 ns; the following products are calculated the every clock cycle.

The first clock approximation was taken to be 3ns and circuit power consumption was found to be 4.77 mW. However, "slack" in the actual circuit was more then 50%. After the gradual decreasing of the clock value from 3 ns to 1.5ns, the power consumption increased in the hyperbolical dependence. Moreover, after code synthesis the circuit area increased as well, however much less comparably than the power consumption. Therefore, it should be taken a smart trade off between clock value, area and power consumed. The pipelined MMP is the key component for the described below Shearer-Composer.

4.3. Shearer – Composer

4.3.1. Shearer- Composer Mathematical Model

In Figure 5, Figure 6, Figure 7 and Figure 52 we showed the volume transformation steps from the original image to the intermediate image coordinate. The Shearer – Composer provides transformation multiplications and compositing into the intermediate image. The Shearer - Composer's mathematical basics are explained in the Chapter 1. The main element of the Shearer, the MMP, was described above in this chapter.

The image data structure we use is a pre-computed run-length encoding of the voxel scan lines. This data structure allows us to skip over transparent voxels during rendering; in other words, we use run-length encoding to determine which voxels of the volume are transparent. This has been done during the pre-processing step that scans through the original volume, classifies each voxel by assigning the opacity, compares the voxel opacity to the threshold, and outputs the run-length encoded representation.

The rendering algorithm traverses the volume slice by slice; at the beginning of the data for each voxel slice the algorithm uses the shear matrix to compute the translation of the slice; then streams through the encoded runs, translates and resamples only the voxels in the non-transparent runs, and composites the resampled voxels into the intermediate image. Therefore, the run-length encoded volume is decoded, sheared, resampled and composited into an image in a single loop without explicitly constructing the sheared volume and eliminating work in the transparent portions of the volume. Run length encoding does not allows random access to the encoded data, in other words, the volume must be encoded from the beginning of the data structure and, therefore, traversed only in the encoding order.

The number of the run length encoded voxels to be processed is read through the corresponding normal registers. The inner processor iterates over slice pixels until the ends of the voxel scanlines are reached. The opaque pixels are skipped, but when a non-opaque pixel is detected, the corresponding voxel is filtered, resampled and composed; then the following pixel is processed. Composition into the intermediate image is done by a summation of the voxels footprints (or voxels contributions) over voxel scanlines.

Parallel Projections are computed as with the following steps:

1. The MMP computes the shear and warp factors of the view transformation matrix.
2. Look up tables for the opacity corrections are included into pre-computing step.
3. The initialization of the intermediate image-compositing into the intermediate image in the scanline order.
4. Image warping and displaying.

Physically, we have pipelined MMP to compute pixels transformations for each slice, skipping transparent voxels. Then the data are written to the Context RAM, and next read by the Classifier and by the Shader for resampling and for opacity correction, computed and then composed by the Composer.

As was mentioned in the Chapter 2, the SWF VRS supports not only parallel projections, but perspective projections as well. The algorithm differs in the way that it requires more matrix multiplications on the first step. Figure 58 shows the shearer-composer structural diagram.

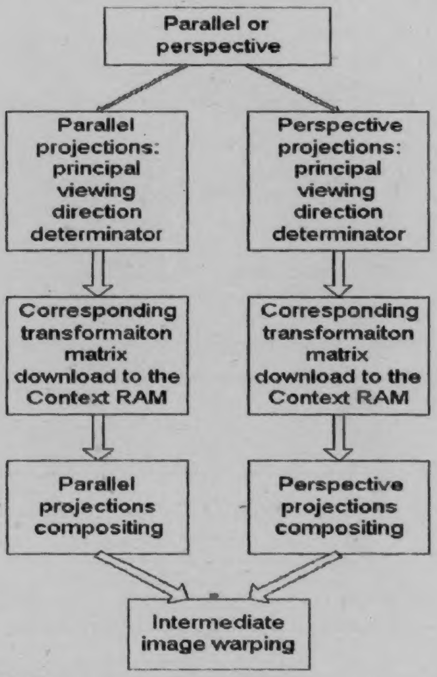

Parallel or perspective
Parallel projections: principal viewing direction determinator
Perspective projections: principal viewing direction determinator
Corresponding transformaiton matrix download to the Context RAM
Corresponding transformaiton matrix download to the Context RAM
Parallel projections compositing
Perspective projections compositing
Intermediate image warping

Figure 58 SWF Composer structural diagram

The Shearer- Composer for both parallel and perspective projections was implemented in VHDL and verified. Compositing is done using a Carry Select adder, described in the following chapter. An FPGA implementation of the Shearer-Composer is described below.

4.3.2. Shearer – Composer Design and Verification

The SWF Shearer-Composer has been designed and implemented in an Altera MAX 7128SLC84-10 FPGA, and its functionality demonstrated. A detailed flow chart is shown in Figure 59. It contains two main parts: Data Path and Controller.

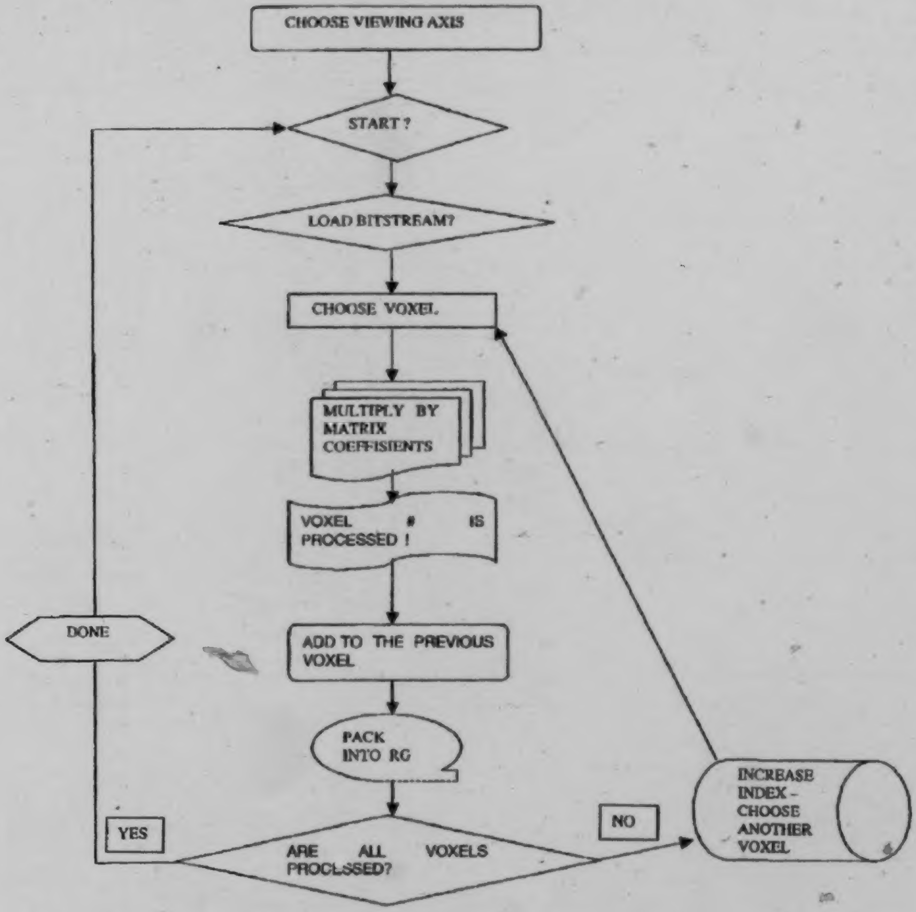

CHOOSE VIEWING AXIS
START ?
LOAD BITSTREAM?
CHOOSE VOXEL
MULTIPLY BY MATRIX COEFFISIENTS
VOXEL # IS PROCESSED !
DONE
ADD TO THE PREVIOUS VOXEL
PACK INTO RG
YES
NO
INCREASE INDEX - CHOOSE ANOTHER VOXEL
ARE ALL VOXELS PROCESSED?

Figure 59 Shearer-composer flow chart

The Data Path structural schematic is given in Figure 60.

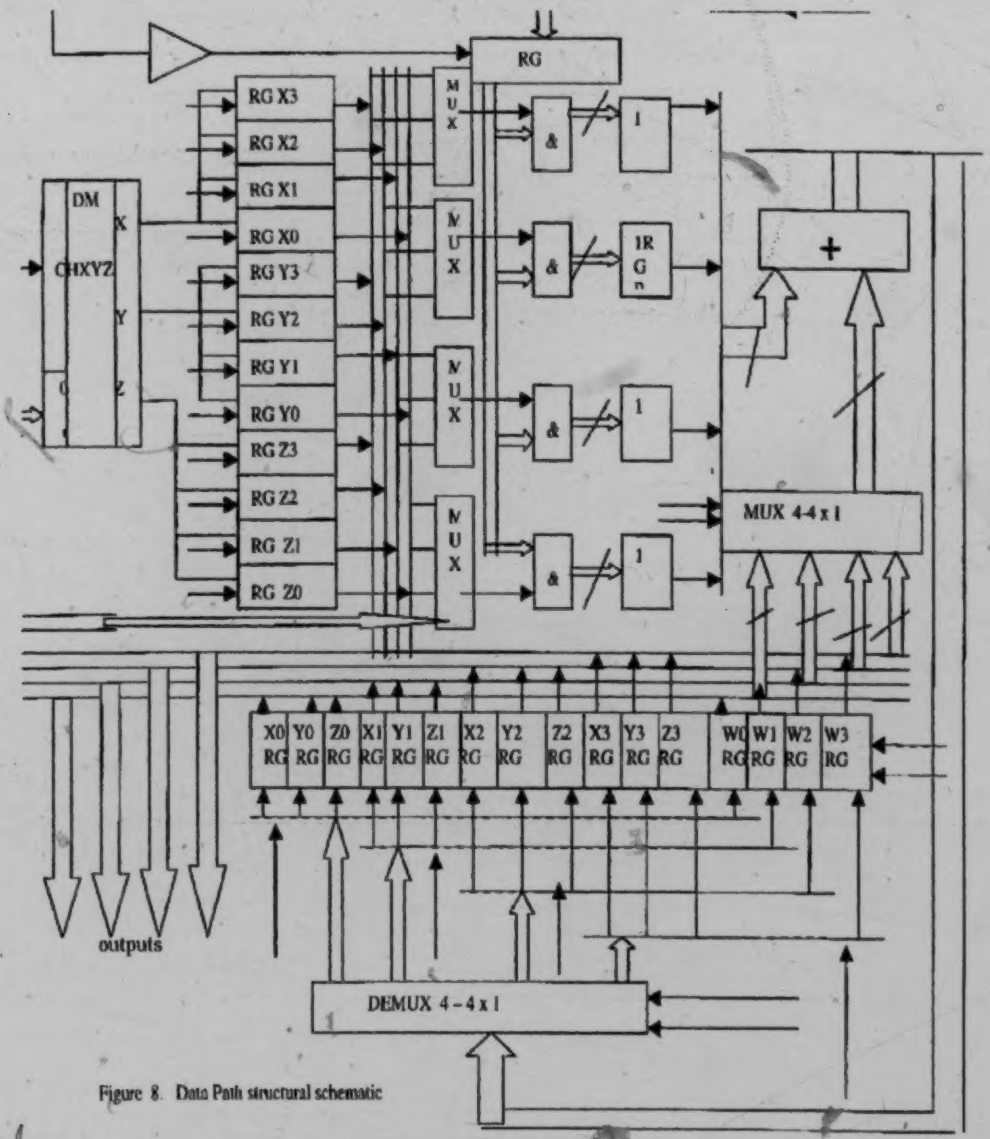


Figure 8. Data Path structural schematic

Figure 60 Data path structural schematic

The Data path consists of two parts: Data path 1 – loading and unpacking the run length encoded bitstream, and transformation into the standard object coordinate system, and Data path 2 - adding the projections to create the intermediate image. Controllers are built separately for each of the data paths. The data voxel of the rendered volume is read from the Shearer Context RAM and loaded into a register.

The Shearer consists of two components "shearorig" and "shearint"; each of which is structurally composed of the components given in the file shcomp.vhd: two data paths and two controllers.

4.3.2.1. Data Path

Pixels are received in a bitstream. In our case, we are limited by the available FPGA, the input array contains 14 bits where 12 bits represent 4 voxels slice of 3D image introduced by X-Y-Z projections as 3-3-3-3 bits and 2 bits "window". The bitstream is loaded into 12 1-bit registers with the CLOCK signal. The first 3 bits are Z-Y-X representation of the first voxel, the last 3 bits are Z-Y-X projections of the fourth voxel. The projections of each of four voxels are separated by the MULTIPLEXER 4-4 x 1 and temporarily stored in a 4-bit register. The transformation matrixes are loaded into the twelve 4-bit registers. Four multiplexers 3N-N select a matrix. The chosen matrix is multiplied by projection (X0, Y0, Z0) of each voxel.

The output of Data path 1 is slice's one voxel in the standard object coordinates. In the Data path 2 this voxel is added to the voxel of the previous slice and stored back in that register, constructing an intermediate image.

4.3.2.2. Control Path

The Controller generates control signals for the data path.

The MUX 4-4 x 1 in Data path 1 is controlled by the signal SM. The perspective axis is chosen by an input signal SDM.

Loading the bitstream is performed by the signal "START'='1' once in the beginning of the bitstream inputting, and the signal 'LD' =1 every time before the bit of information is inputted. The output is a 16- bit intermediate voxel.

The Controller controls the timing of the whole system based on the CLOCK signal. A Moore State Machine with asynchronous reset has been used to build CONTROLLER.

4.3.2.3. Testing strategy for Shearer - Composer

The following aspects of Shearing circuit work were tested:

- Random test vectors set,
- "Chess" code (the test has the static and dynamic errors detection efficiency equal to "running one", but much less timing consuming),
- Choosing a different viewing axis,
- Boundary cases:
  - When image is totally solid (all X-Y-Z =1),
  - When the image is transparent (all 0).

Each structural component of the Shearer –Composer was tested separately and exhaustively. For the Controllers the different situations were modulated, and the component's behavior was analyzed. For the Data Path, the input signals and control signals expected from Controllers were modulated testing outputs. Then, the parts of the Shearer-Composer, responsible for standard and for intermediate images were tested separately. As the last step, the Shearer-Composer, composed structurally from both mentioned above parts, was tested as a whole unit.

Overall, 32 test benches were applied to test all signals and the Shearer-Composer functionality. Below two of them are shown:

- Figure 61 shows how two non-transparent voxels are added: previous voxel value of "1100" and the following "0011", the result "1111"

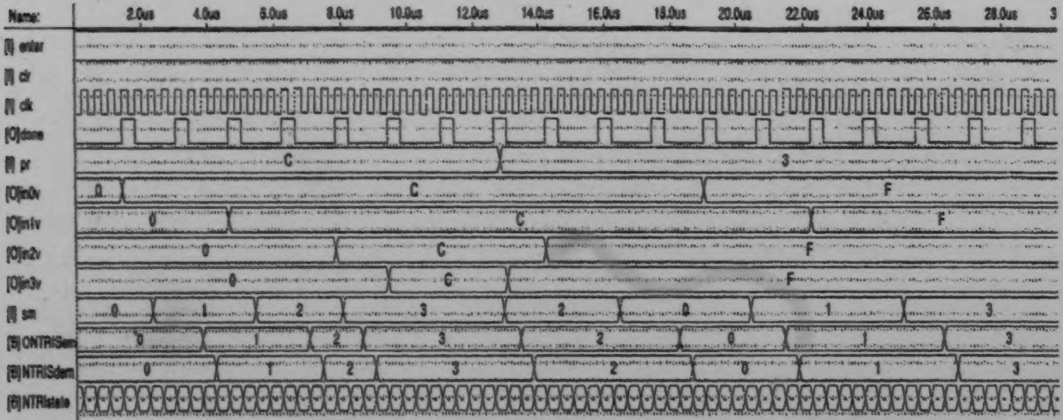


Figure 61 Composer functional simulation: non-transparent voxels' compositing

- Figure 62 shows that transparent voxel value "0000" has been detected and skipped from processing. The previous voxel value in the intermediate image was "1111" and remained so for the future processing.

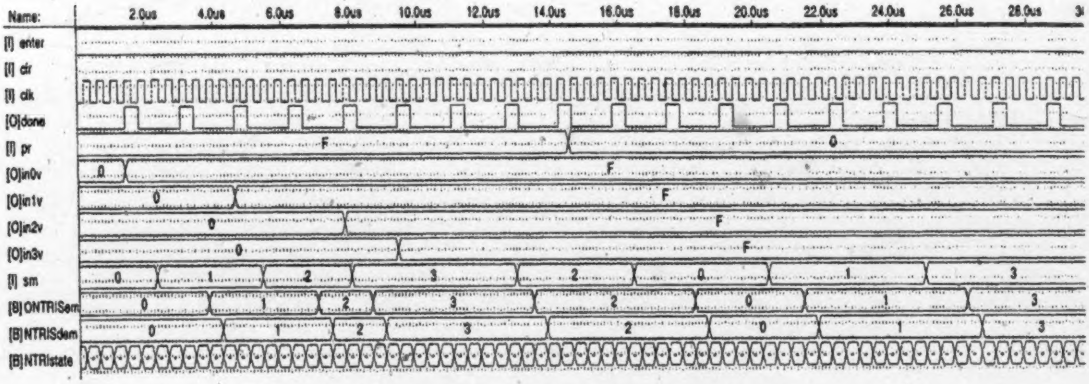


Figure 62 Transparent voxel skipping

Thus, the composer in intermediate image kept the same value of the voxel after transparent voxel and changed the value after non-transparent voxel.

4.3.2.4. Implementation in MAX 7128SLC84-10 FPGA

To implement the design of the shearing unit in a 7128SLC84-10 FPGA the following must be completed:

1. Switch debouncing (push buttons and switches);

2. Bitstream loading is performed as:

   - push START button
   - set switch button to the corresponding the bit -value position
   - push LD button – the bit of the bitstream is loaded into bit –registers
   - set switch button to the next bit -value position
   - push LD button – the bit of the bitstream is loaded into bit –registers and previous load moves to the next bit register

3. Loading is repeated 12 times until all registers are loaded. The principal viewing axis can be chosen by signal SDM. If SDM = 00 then the result is 0; if SDM=11 then the matrix X-axis is chosen; if SDM=01 – matrix Y; if SDM=10 then matrix Z.

Shearer- composer code compilations results are given below:

```
**** Project compilation was successful

SHEARER

** DEVICE SUMMARY **

Chip/                  Input  Output  Bidir        Shareable

POF      Device         Pins   Pins    Pins   LCs  Expanders  % Utilized

shearer  EPM7128SLC84-10 7      23      0     120     3         93 %

User Pins:               7      23      0

§

Project Information                          c:\natalie\shearer.rpt

** PROJECT COMPILATION MESSAGES **

** AUTO GLOBAL SIGNALS

**INFO: Signal 'clk' chosen for auto global Clock

INFO: Signal 'clr' chosen for auto global Clear

** STATE MACHINE ASSIGNMENTS **

|shearint:CONTR|shcontrol:CONTR|state: MACHINE

      OF BITS ( |shearint:CONTR|shcontrol:CONTR|state~4,

                |shearint:CONTR|shcontrol:CONTR|state~3,

                |shearint:CONTR|shcontrol:CONTR|state~2,

                |shearint:CONTR|shcontrol:CONTR|state~1 )

      WITH STATES (

                 s0 = B"0000",

                 s1 = B"0110",

                 s2 = B"0010",

                 s3 = B"0100",

                 s4 = B"0111",
                s5 = B"1000",
                s6 = B"0101",
                s7 = B"1010",
                s8 = B"1001",
                s9 = B"0011",
                s10 = B"0001");
|shearorg:sh_data_int|shcontorig:sh_data_org|state: MACHINE
    OF BITS ( |shearorg:sh_data_int|shcontorig:sh_data_org|state~3,
              |shearorg:sh_data_int|shcontorig:sh_data_org|state~2,
              |shearorg:sh_data_int|shcontorig:sh_data_org|state~1  )
    WITH STATES (
                s0 = B"000",
                s1 = B"010",
                s2 = B"110",
                s3 = B"100",
                s4 = B"101",
                s5 = B"011",
                s6 = B"001");
Timing SNF Extractor              = on
Optimize Timing SNF               = on
Fitter Settings                   = ADVANCED
```

4.4. Shading Processor

4.4.1. Shading Technique

In this section we discuss the shading and illumination models. In the volume rendering, photo realism is not a goal. A volume data set is a representation of an object and its internals. Shading is used for the better understanding of the data set- spatial cues and structure. In Figure 63 the shading stage is defined in the rendering pipeline.

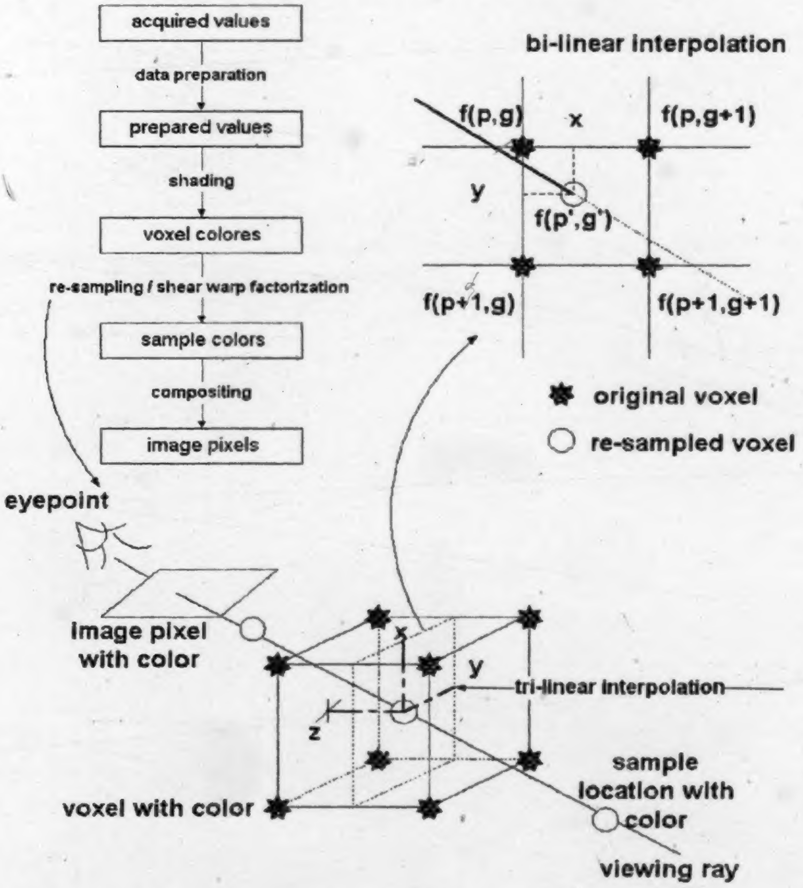


Figure 63 Shading stage in volume rendering pipeline

Constant (Uniform)

The simplest possible shader: includes only the surface color, which is constant across the surface. This is because the surface normal, reflectance and lighting information are not calculated at all. This model is used with texture mapped surfaces when the texture image is preferred to be reproduced on the object as close to the original color as possible.

Lambert

Calculates the surface normal by interpolating between normals of two adjacent polygon normals, resulting in a smoothly shaded object. The surface reflectance is not incorporated, which yields Lambert shader suitable for matte surfaces with unpolished chalk-like look. This shader is based on the Lambert's cosine law discovered by Johan Lambert, a sixteenth-century astronomer and physicist. Lambert's cosine law simply states that the intensity of light on a surface is proportional to the angle at which the light hits the surface [12].

Gouraud

Calculates the surface normal by *linearly* interpolating between normals of adjacent polygons. A Gouraud shader [12] is a simplified version of Lambert shader and is especially suitable for real-time rendering on graphics hardware, i.e *hardware rendering*. It was developed by Henri Gouraud in 1971.

Phong

Calculates the normal separately for every pixel on the surface and also processes the relation between normal, direction of the light source and the direction of the camera's point of view. This method gives a much better surface curvature. Phong shading is the most suitable for plastic materials. Though much more computationally expensive than Lambert or Gouraud, Phong is the most popular shading method today. Developed by Bui Tuong Phong [12] [76].

Blinn

This algorithm calculates the surface normal very much like Phong, except that the shape of the specular highlight reflects the actual lighting more accurately. It was developed by James (Jim) Blinn [73].

Although it was mentioned that Gouraud shading model is the best suitable for the graphic hardware since it is less computationally intensive, for our purpose we have chosen the Phong shading approach because it provides a more accurate way of shading a polygon since the illumination model is applied to every point on that polygon.

Therefore, the computational intensity became the problem to be solved. Below the mathematical solution of this issue is presented.

4.4.1.1. Phong Shading

Phong's method [12] determines the intensity of each point using an approximate surface normal that is linearly interpolated from the true surface normals specified at the vertices. Real-time Phong shading [76] has always been a goal for 3D coders, however the massive amount of calculations involved has hampered progress in this area. The Phong shading is like the Gouraud shading based on an interpolation algorithm, but the interpolation is made by vectors. It uses the normals at each pixel, the average normals at each vertex, and interpolates vectorally along the edges between to vertex, and then interpolates the same way between the edges along the scan-line.

4.4.1.1.1. The Phong illumination model

This is given by the following algorithm:

color = ambient + diffuse *(cos x) + specular *(cos x)^n

Simply, the ambient component defines the color of an object when no light is directly striking it, the diffuse component defines the color of the object, and the specular component defines the color of the highlight made when light strikes the object directly. The angle x should have a range of 0 to 90 degrees, since that is the range of angles possible when light intersects a visible plane. The power n in the specular component defines certain attributes about the material, the greater the power n, the shinier the material will appear to be (i.e. the specular highlight will be smaller and brighter as n increases).

4.4.1.1.2. The Phong shading model

The idea behind Phong shading is to find the exact color value of each pixel. Therefore, in its most classical form, the Phong shading model is as follows:

1) to determine a normal vector at each vertex of a polygon, the same normal vector used in gouraud shading
2) to interpolate normal vectors along the edges of the polygon

3) to interpolate normal vectors across each scanline, so we have one normal vector for each pixel in the polygon

4) to determine the color at each pixel using the dot product of the normal vector at that pixel and the light vector (the same method used in Gouraud and Lambert shading). Since the interpolated normals are not of constant length, this step requires a square root to find the length of the normal.

Classical Phong shading model is shown below in Figure 64 [12], and described by Equations 37. This model gives a better representation of specular lights.

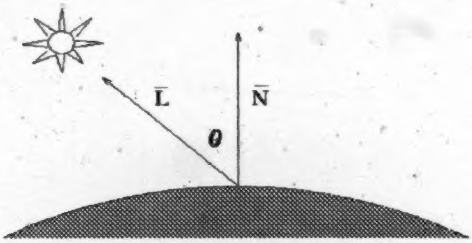


Figure 64 Phong Shading parameters

$$I = IaKa + Ii(Kd(LiN) + Ks(LiR)\hat{}\ n) \quad \textbf{Equation 37}$$

where the first term is ambient intensity, and the second term is diffuse intensity.

One of the hardware implementations has been proposed by Juskiw [39] for Bela Volume rendering project. The structural schematic is shown in Figure 65. This implements the classical Phong equation given in Figure 64.

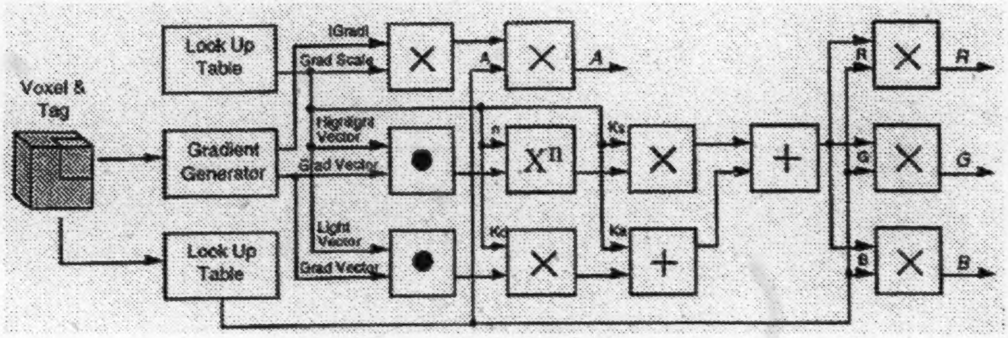


Figure 65 Bela VRS shader schematic

Table 8 Arithmetic operations for Phong shading with a single light source (n is a specular exponent)

Math operation	Ambient and diffuse shading	Full Phong equation
Additions	7	15 + n
Multiplications	11	19 + n
Divisions	1	2
Square roots	2	3

Table 8 shows the results obtained by Snip [40][41] after brute-force implementation of Bela shader. Phong shading required a tremendous amount of interpolation and one square root per pixel. Even with lookup tables for the square root function, there was no way that this algorithm would be fast enough to use in real-time. To speed up calculations and reduce their amount some mathematical derivations has to be made.

4.4.1.2. Mathematical Derivations

The colour Phong shading model is:

$$Ir = IaKar + Ii(Kdr(LiN) + Ks(LiR)^\wedge n) \quad \textbf{Equation 38}$$

$$Ig = IaKag + Ii(Kdg(LiN) + Ks(LiR)^\wedge n) \quad \textbf{Equation 39}$$

$$Ib = IaKab + Ii(Kdb(LiN) + Ks(LiR)^\wedge n) \quad \textbf{Equation 40}$$

where the colour is expressed in terms of (Ir, Ig, Ib).

Linear interpolation, defined by the following equation,

$$N(x, y) = Ax + By + C \quad \textbf{Equation 41}$$

is chosen to interpolate the normal across the polygon. Interpolation for successive values of x and y and evaluating the illumination model requires 7 additions, 6 multiplications, 1 division and 1 square root per pixel.

Phong shading can be implemented more efficiently by combining the interpolation and reflection equations.

$$I(diffuse)(x,y) = \frac{L}{|L|} \times \frac{Ax+By+C}{|L||Ax+By+C|} \qquad \textbf{Equation 42}$$

Which can be rewritten as

$$I(diffuse)(x,y) = \frac{L \times Ax + L \times By + L \times C}{|L||L \times Ax + L \times By + L \times C|} \qquad \textbf{Equation 43}$$

Performing the indicated dot products and expanding the vector magnitude yields:

$$I(diffuse)(x,y) = \frac{ax+by+c}{\sqrt{(dx^2+exy+fy^2+gx+hy+i)}} \qquad \textbf{Equation 44}$$

where $a = \frac{L \times A}{|L|}$; $b = \frac{L \times B}{|L|}$; $c = \frac{L \times C}{|L|}$; $d = A \times A$;

$e = 2A \times B$; $f = B \times B$; $g = 2A \times C$; $h = 2B \times C$ and $i = C \times C$

Using forward differences, this form can be evaluated for successive values of x and y with 4 multiplications, 3 additions, 1 division and 1 square root per pixel. This is a substantial savings over Phong's formulation but the division and square root are still too expensive for real-time use.

Therefore, Phong shading is achieved with only a little more computation per pixel than is required for Gouraud shading. Again, after some optimizations we can eliminate one multiply and replace the square root with a lookup table, but three multiplications and one division per pixel are still far too slow to be used in real-time.

The following considerations will bring us to the continuous Phong model optimization. Phong shading is the most correct method of shading because it is not an approximation since distinct colors are determined for each pixel. It states that the intensity of light at a given point is given by the dot product of the light and normal vectors divided by the product of the magnitudes of the two vectors. To graph the intensity of light striking flat shaded planes, it is obvious that they roughly form a cosine curve, since the color values at certain points are determined by the cosine of the angle between two vectors. Actually, these values follow the cosine curve exactly, because the intensity of each pixel was calculated using a dot product, which eventually yields the cosine of the angle between the plane at that pixel and the light vector.

We can conclude that shadings calculated using the Phong model follow the cosine curve, because the dot product between the normal vector at each pixel and the light vector can be simplified to a function involving cos(a). Knowing this, another task arises- to find a fast way to evaluate that function.

The Taylor expression will approximate this as:

$$\cos(x) = x - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + (-1)^{n-1} \times \frac{x^{2n}}{(2n)!} \qquad \textbf{Equation 45}$$

More terms give more accuracy but reduce speed. For the maximum demand of a 16 bit representation (in our case), it will be sufficient enough to calculate only 2 products of the Taylor series, this will give an accurate approximation from x = 0 to x = 90, which are the limits of angle between the light and visible facets. Therefore,

$$\cos(x) = x - \frac{x^2}{2} \qquad \textbf{Equation 46}$$

The result shows that this interpolation is not linear! Thus, we came back to the quite complicated shading Phong model.

4.4.1.2.1. Angle approximation?

The angle between the light vector and the normal vector at each vertex of a plane are changing in a linear fashion from one vertex to the next and from one edge of the plane to the next across each scanline. The Phong shading model calls for normal vectors to be linearly interpolated from vertex to vertex and from edge to edge. The actual coordinates of these normals do not matter, only the angle between the vector defined by these coordinates and the light vector. Why interpolate three coordinates per pixel when they are just an intermediate step in finding the angle between two vectors?

So, angular interpolation eliminates the interpolation of a whole vector. It also eliminates the dot product, which was previously needed to find the cosine of the angle between the normals and the light. Without the dot product or vector interpolation, there is nothing left of the traditional method of Phong shading. All that needs to be done is to interpolate angles across the plane, and to look up the cosine of those angles in a lookup table; the only possible values for the smallest angle between a normal vector and a light vector are 0 to 90 degrees.

Now, instead of interpolating colors across the plane, we need to interpolate angles instead, and then assign a pixel its correct color. The color calculations for each pixel are the same as those for Lambert and Gouraud shading. We multiply the cosine by the number of colors in the gradated palette range and add the result to the base color of the range. The angle between the light and the normal vectors at each vertex can be accomplished by the inverse cosine function. By taking the inverse cosine of the cosine, we get the angle as a result.

Just one limitation was found. Angular interpolation is correct in 99% of all possible cases. The only case when it is not correct is when the angles at any two or more vertices are equal. Interpolated vector Phong shading will display a specular highlight inside the plane in such a case, but interpolated angle Phong shading will render the plane in a solid color because the difference between the angles at the vertices is 0, there will be no change in the angle across the plane.

For future work the only VLSI implementation of the shader is left.

4.5. Classifier

The classification group refers to mapping methods that map data values to different materials or surfaces. It includes transfer functions and surface classification functions. The data values are typically mapped to color (red, green, blue) and opacity values. In the classification stage, Figure 66, the color and opacity are signed to voxel.

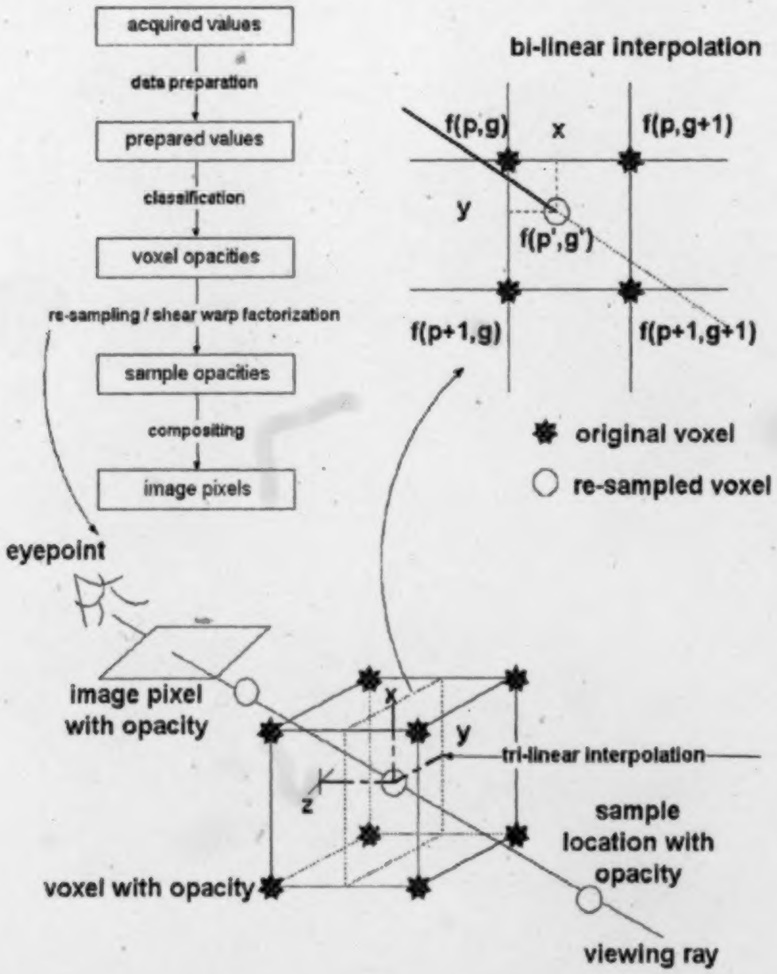


Figure 66 Opacity weighted color interpolation pipelining

In the proposed by Lacroute shear- warp factorization algorithm it was done separately. This is illustrated in Figure 67; in the left figure color and opacity are interpolated separately, in the right figure - opacity weighted color interpolation occurs.

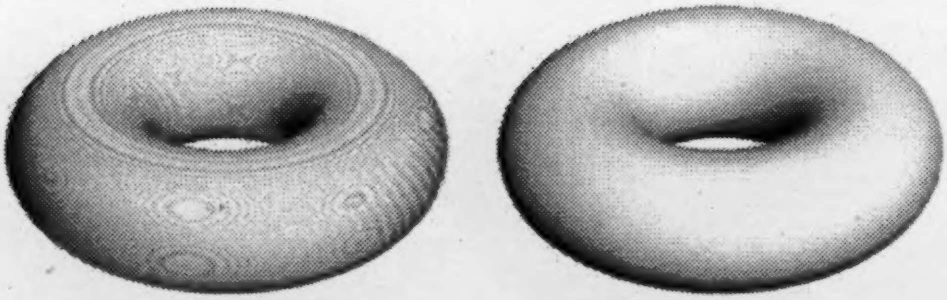


Figure 67 Artifacts of separate interpolation of color and opacity

To avoid the illustrated artifacts in the classification stage the algorithm shown in Figure 66 will be used in future work. The Classifier will be implemented in C, and this is a part of the future work.

4.5.1. Chapter Summary

In this chapter we described SWF system level components: Sobel gradient processor, Shearer - Composer, and Matrix Multiplication processor. This chapter includes also the explanations of the Shader mathematical model, and shows the artifacts found during the classification stage in the developed by Lacroute Shear-Warp due to algorithmic error.

The designs of sub-components, which form the package for the hierarchically higher components (described in this chapter), are presented in the following chapter. They are defined as circuit level components.

Chapter 5

5. Circuit Design and Verification

This chapter describes design and verification of the Inner Product Processor, its components, Booth Multiplier, Wallace Reduction tree and two-operand 16-bit Adder, and high speed and low power Non Full Swing Complementary Pass Transistor Logic (NFS CPL) used for the sub-components design. The sub-components, described below, are the basic blocks, which form the complex components, presented in the previous chapter.

5.1. Inner Product Processor

5.1.1. Applicatory Aspects

The inner or dot or scalar product evaluation is the basic operation in higher mathematical spaces such as vectors and matrices and in almost all digital signal-processing algorithms as well. The evaluation of the dot product is an important calculation in computer graphics applications. Furthermore, it is important for image processing and volume rendering since both of these techniques borrow from conventional computer graphics. The dot product processor is an integral part of a multifunctional processor for a Volume Imaging System (VIS), which requires real–time performance with very low power dissipation. The massive amount of computations required for a full-frame interactive VIS system dictates that the inner product be computed in a minimum of 8-10 ns for 16-bit fixed-point signed operands. However, such optimization for high-speed performance usually means complex circuits and excessive power consumption resulting in increased system overhead and reduced reliability. Therefore, this design has to consider trade-offs between speed and power consumption. The inner product can be given by the formula below:

$$A \bullet B = \overrightarrow{a1} \times \overrightarrow{b1} + \overrightarrow{a2} \times \overrightarrow{b2} + \ldots + \overrightarrow{aN} \times \overrightarrow{bN} = \sum_{i-1}^{N} \overrightarrow{ai} \times \overrightarrow{bi} \quad \textbf{Equation 47}$$

or $$A \bullet B = |A| \times |B| \times \cos\theta$$ **Equation 48**

The inner product has other applications. The most important of these pertains to finding the angle θ between two vectors A and B (or between two intersecting lines) that is commonly used in computer graphics applications:

$$\cos\theta = Dot\_product(a,b)/(lenght(a) \times lenght(b))$$ **Equation 49**

One of the most important applications of the inner product is for lighting and shading computations such as in the Phong Shading Illumination model.

The scalar product is also an important part of the clipping algorithm. This gives the value of vector's components compared to the eigenvectors..

Finally, the dot product can be used in sampling ray vectors and statistical correlation for perspective adjustment between eye distance and the number of volume samples.

5.1.2. IPP Architecture Based on the Merged Arithmetic

Merged Arithmetic (MA) was developed for signal processing applications. The main idea of this technique involves synthesizing discrete arithmetic operations by decomposing them fully or partially. The "level" of decomposition defines the MA technique [13]: Fully Merged or Partially Merged. The obtained structure usually provides faster computation, reduced transistor count and hence reduced power dissipation. One of the most important applications of MA is inner product computation. In the Fully Merged Architecture (FMA), the boundaries between multiplication and addition are completely dissolved into a single functional circuit. In Partially Merged Arithmetic (PMA), separate multipliers perform the column compression until the bit product matrix is reduced to two equivalent rows, which are then summed by a multi-input adder just once.

Since the speed of computation is critical for real-time image processing, a parallel type multiplier architecture was adopted over a bit-serial implementation, although the latter consumes less power. A parallel multiplier can be divided into three main blocks: the partial-product generator (PPG), requiring a shift and add for Booth-Encoding, the adder block to reduce the partial products to two rows and finally, a two operand adder to produce the final result. Furthermore, such architecture is amendable for pipelining, which increases throughput while having the drawback of increased latency. Pipeline registers can be placed after the partial product generators, at each level of the Wallace Tree and before the final two-operand adder.

5.1.3. Modified Booth Encoder

Modified Booth Encoding [9] is used to reduce the number of partial products to be summed. Modified Booth encoding allows the grouping of a greater number of bits compared to Booth encoding with the number of the partial products reduced by 50%.

5.1.3.1. Encoder

Since the encoder for each set of input bits must drive heavy loads (fanout of 9 or 10) large static CMOS inverters are connected to the outputs. To avoid short-circuit dissipation, the cell before the static CMOS inverter must be of a full swing type. The encoder generates output signals resulting from (Y) adjacent ternary bits in the multiplier. These outputs are: X, 2X and NEG. Active HIGH on one or two of these outputs serve as control signals to the Partial Products Generator as to which partial products are to be generated. A schematic of the Booth encoder cell is shown in Figure 68.

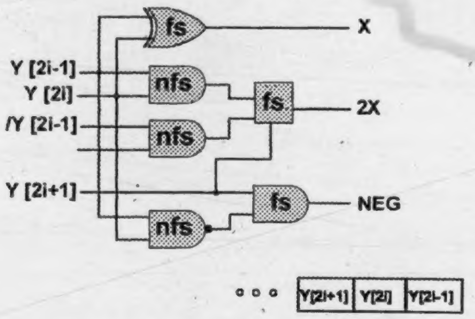


Figure 68 Booth encoder basic cell

5.1.3.2. Partial Product Generator

The PPG generates bits of the partial product defined by the encoding signals:

either X = multiplicand, or 2X = one bit left shifted multiplicand, if NEG = active HIGH, the ones complement of X or 2X, or produces a “zero” partial product if all the three adjacent bits in the encoding multiplicand are “0” or “1”. A schematic of the PPG cell is shown in Figure 69.

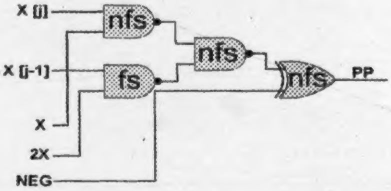


Figure 69 Partial product generator cell

5.1.3.3. Two’s Complement Adder

To complete the final partial product, a two’s complement adder (2C adder) is needed to change all of the inverted partial products into two’s complement form. The basic cell is based on a “half” adder. The “half –2C ” adder is based on the following derivations:

$S = A \oplus B \oplus Cin$ **Equation 50**

$Cout = A * B + Cin * (A + B)$ **Equation 51**

Assuming that B = “0” and $C_{in}(0)$ =”1,” we derive the following:

$S = A \oplus Cin$ **Equation 52**

$Cout = Cin * A$ **Equation 53**

The schematic of the 2C adder is shown in Figure 70.

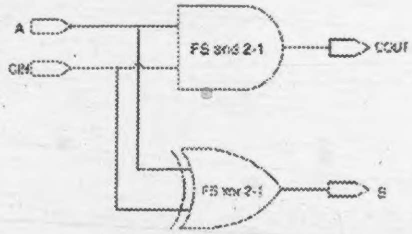


Figure 70 Two’s compliment adder cell

Therefore, C_{out} is propagated to the C_{in} of the next most significant bit. To reduce the rippling effect, that this propagation causes, a Carry–Select Adder architecture was adopted. The carry-selection is divided into three blocks of width 2 – 3 – 3 bits. Schematic of the 2-3-3 bits two's compliment carry-select adder is shown in Figure 71.

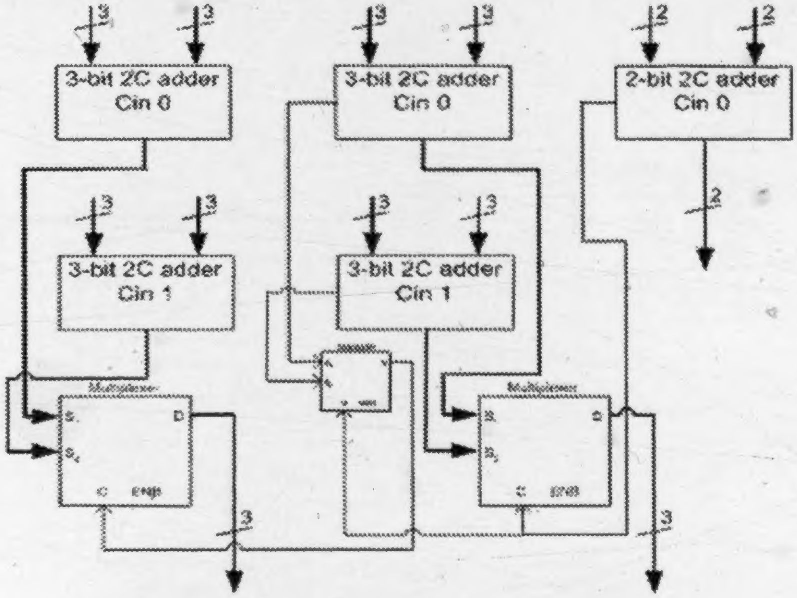


Figure 71 Two's compliment carry select adder

Thus, the maximum rippling delay is 3 gates. Full Swing (FS) cells are used in the "half" adder. Insertion of buffers between 2C adder cells is not needed because the full-swing C_{in} signal connects to the poly gates of pass–transistors. It should be pointed out that the sign bit for a partial product does not take part in two's complement adder addition. A single partial product only connects to the input of one 4-2 compressor. However, large driver inverters are needed at the outputs because the wiring capacitances from these outputs to the inputs of the Wallace tree are significant, and hence affect the robustness of a design. According to [22], when a single gate in the PPG has to drive a fanout of, at least, four similar gates in 4-2 compressor of the Wallace tree a single inverter should be used.

5.1.3.4. Sign Extension Simplification

Sign bit extension can be simplified significantly by using the following algorithm with two additional bits before the sign bit:

For the first partial product

$$PPn+2 = PPn+1 = PPn$$ **Equation 54**

For the second partial product

$$PPn+2 = PPn+1 = 1 \text{ if } PPn = 0$$ **Equation 55**

$$PPn+2 = \overline{PPn+1} = 1 \text{ if } PPn = 1$$ **Equation 56**

F (flag) is generated by the previous partial product to the next one:

$$Fj+1 = Fj + PPn,j$$ **Equation 57**

where- PPn,j is the sign bit of the j-th partial product.

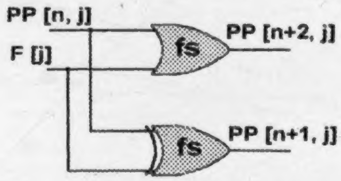


Figure 72 Sign extension simplification cell

The sign extension simplification circuit, shown in Figure 72, is responsible for generating the 10-th and 11-th bit for each partial product, except for the most significant partial product, PP3, where the 11-th bit is ignored. It employs FS cells and buffered outputs to drive the wiring capacitance to the 4-2 compressors as in the 2C adder.

The rules for generating the sign extension bits are as follow: the 9-th bit, the sign bit of the partial product is used as the basis for generating bits 10 and 11. The 9-th partial product bit, as we recall, is generated in the Booth Encoder complying with the same rules as for the 9-th, except that it does not take part in two's complement addition, which cannot be applied to the sign bit. Again buffering is needed on the outputs of the sign extender to drive the wiring capacitance to the compressor inputs.

5.1.3.5. 3-D Booth Encoder

The entire Booth Encoder for two 8-bit signed operands produces four 10/11 bit partial products with full-swing outputs. Since we are dealing with 3D vectors, all 12 partial products are generated in parallel. The Booth Encoder outputs for vectors A (Xa, Ya, Za) and B (Xb, Yb, Zb) are named as:

PPX0<0:10>, PPX1<0:10>, PPX2<0:10>, PPX3<0:9> for X – axis;

PPY0<0:10>, PPY1<0:10>, PPY2<0:10>, PPY3<0:9> for Y – axis;

PPZ0<0:10>, PPZ1<0:10>, PPZ2<0:10>, PPZ3<0:9> for Z – axis.

Thus, the modified Booth Encoder reduced the number of partial products by half.

One of the approaches lies in partial product addition in using an adder array, called a Braun Multiplier [15], which has a regular structure and, therefore, is highly amendable to a VLSI layout. However, it suffers from poor speed performance and dissipates unnecessary power from spurious transitions. The Wallace Tree reduction algorithm is recognized to be the fastest according to Bellaouar [15] and is chosen for the implementation.

5.1.4. Wallace Reduction Tree

A Wallace Tree [16] is used for the column compression. For larger multipliers, the Wallace Tree technique has proven to provide good speed performance due to its high level of parallelism. However, tree adders based on full adders or 3-2 compressors, as originally described in Wallace's paper [16], result in highly irregular layouts, which substantially increase the power dissipation of multiplier's architecture. This is because the interconnect delay, and hence the speed and power dissipation, has been shown to be directly related to the circuit size and shape.

Multiple input compressors that can sum up several partial products simultaneously are often used to circumvent this problem. A (n, m) compressor or counter is really just a combinational circuit with n inputs and m outputs. The outputs are a binary coding of the sum of the input bits. Therefore, for certain bit positions, a maximum of nine bits need to be added. In our case, two stages of 4-2 compressors are therefore necessary to reduce the partial products as shown in Figure 73.

Such architecture has been shown to be highly regular and amendable to a VLSI implementation. The Booth encoding and decoding process introduces extra circuitry at the inputs and outputs of the inner product processor with a resulting reduction in the number of partial product rows.

The Wallace Tree lends much better speed performance due to the high level of parallelism, constructed using multiple input compressors that can sum several partial products concurrently. The Wallace Tree technique is combined with the Modified Booth Encoding technique to reduce the number of partial product and the Fully Merged Arithmetic technique to decrease the number of needed compressors, and hence to reduce power consumption because of lower transistor count.

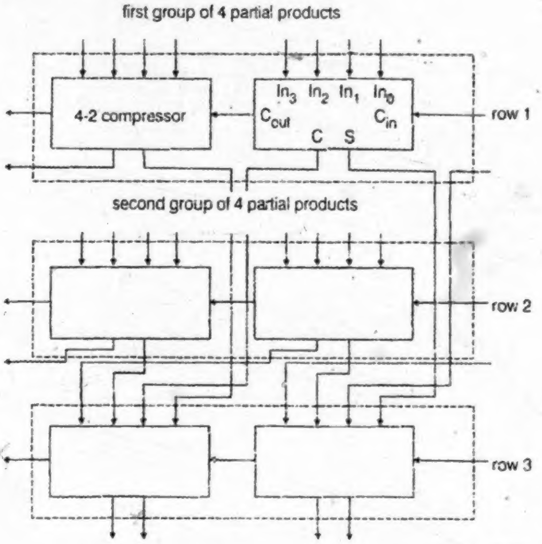

first group of 4 partial products
4-2 compressor
In3 In2 In1 In0
Cout
Cin
C S
row 1
second group of 4 partial products
row 2
row 3

Figure 73 Section of Wallace Tree using 4-2 Compressors

Of the available multi-input compressors, it was decided that the most optimal tradeoff between speed and power dissipation, eventually defined by the number of gates in compressors and by lateral or vertical propagation, would involve the use of 4-2 compressors in the reduction tree. The regular layout that can be achieved with these compressors is important since, the speed and power dissipation are not determined only by the cell design but also by the length of the wiring interconnections. It requires three stages of 4-2 compressors to reduce two 3D 8-bits vectors to two terms that can be summed using a two operands adder. The reduction of partial products in the Wallace Tree is shown in Figure 74, and the concept is explained in the following sections.

Figure 74 shows three stages of 4-2 compressors. The numbers of 4-2 compressors in each stage are given in the top rows. Four dots in an oval introduce the inputs of 4-2 compressors, and the way in which partial products are compressed. Two outputs of the each 4-2 compressor propagate to the next stage, where, in turn, are compressed with other two inputs, and so on. Connectors between 4-2 compressors introduce Carry Out – Carry In propagation within stage. Inputs, which are left uncompressed within the stage, propagate to the next stage.

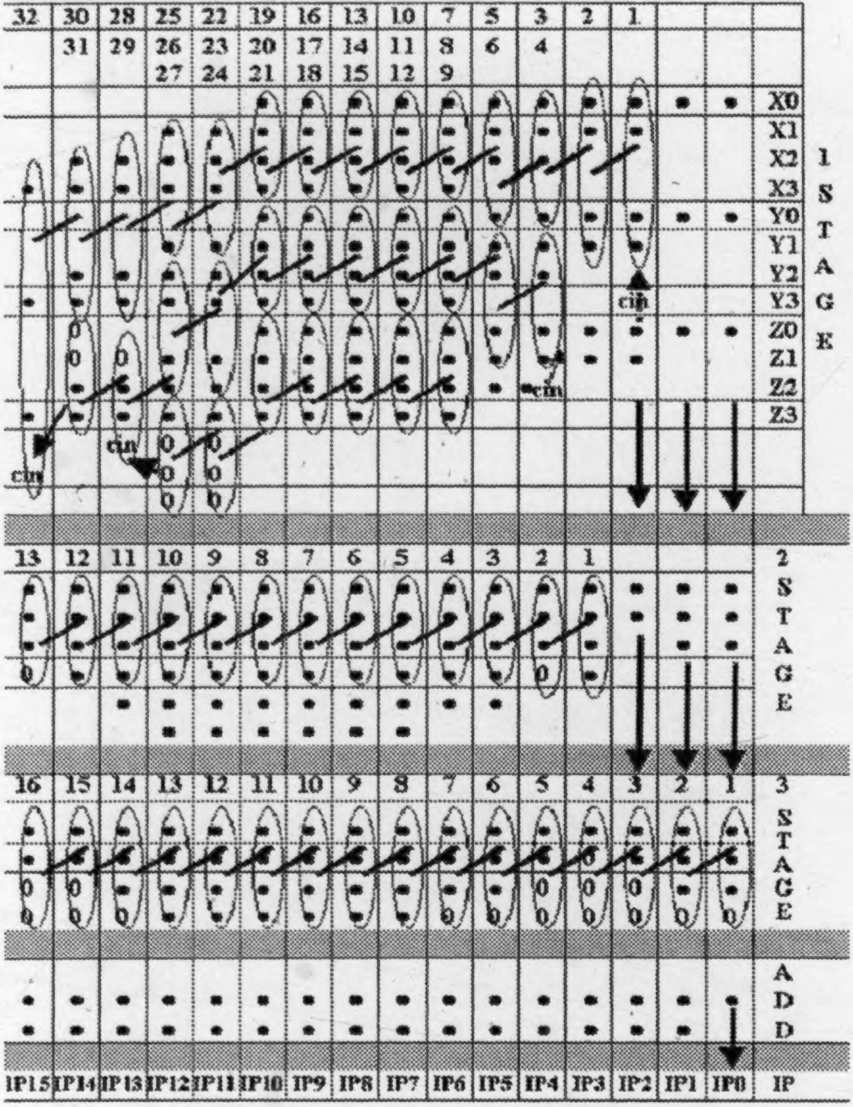


Figure 74 Wallace tree bit product reduction

5.1.4.1. First Stage/Level 1

The first stage of the Wallace tree has the most irregular structure, as shown in Figure 74, that might result in irregular layout. It contains 32 4-2 compressors. The inputs to these compressors are FS partial products driven by large buffers to circumvent the wiring delay from the Booth Encoder to the compressor inputs.

5.1.4.2. Second Stage/Level 2

The second stage inputs are FS Sum and Non Full Swing (NFS) Carry outputs from the corresponding first stage 4-2 compressors. For proper connection In0 and In2 inputs are connected to the FS Sum outputs from the first stage; In1 and In3 are connected to NFS Carry outputs. This stage contains 13 compressors and has a very regular structure that is highly amendable to VLSI implementation.

5.1.4.3. Third Stage/Level 3

This stage has the most regular structure. The third stage includes 16 4-2 compressors. Again inputs, Sum and Carry from the second stage, are connected to appropriate FS - NFS nodes so that signal degradation does not degrade below one threshold voltage drop. Outputs of this stage are 2N = 16 bits of Sum<15:0> and Carry <16:1>, which create the two 15 bit operands to be added. It should be pointed out that the Carry 4-2 output is NFS. This fact will be accounted for in the design of the final two-operand adder.

5.1.5. Adder

It is important that the final two-operand adder performs the addition as fast as possible to provide the output of the inner product operation. A Carry – Free Adder has been recently proposed by Iijoo [19]. It has only 3 gate delays for all bits in the sum with additional minor delays for cascaded multiplexer selection. This adder accepts two redundant binary numbers and produces a single redundant binary number. Therefore, the reduction rate is 2 to 1. The negation of a redundant binary number is obtained simply by converting non-zero positive digits into negative ones or vice-versa without a carry propagating addition. Two bits (X*, X**), according to the following encoding rule, shown in the Table 9, represent a redundant binary digit (X).

Table 9 Redundant Binary Addition Encoding Rule

X	X*	X**
0	0	0
1	0	1
0	1	1

Figure 75 shows an example for an 8-bit redundant binary addition; X and Y are n-digit redundant binary integers. I-Sum and I-Cin are intermediate Sum and Carry-in. Final Sum (F-Sum), which is obtained by adding I-Sum and I-Cin. Addition is carried out using the adding rule given in Table 10. Note that there is no carry generation in the addition of I-Sum and I-Cin to satisfy a carry-free condition and the LSB of I-Cin is set to logic zero.

Augend (X)		1 0 1 1 0 0 1 1(-77)	
Addend (Y)	+	0 1 1 0 0 1 0 1(101)	
I-Sum		1 1 0 1 0 1 1 0	Init. Cin
I-Cin		1 1 0 0 1 1 1 0	Cout from LSB
F-Sum		0 0 0 1 1 0 0 0 (24)	

Figure 75 Redundant Binary Addition

Table 10 Adding Rule

Xi, Yi	0	0		0		1-	1	1
	0	1		1-		1-	1	1-
I-Sum	0	1-	1	1	1-	0	0	0
I-Cin	-		0		0	-	-	-
		1	1-	1-	1			
Cout	0	1	0	1-	0	1-	1	0

In the above table, i .(0,1,.....n-1), n is the number of digits in the adder and "-" is the don't care condition, which can be substituted by "0", "1" or "1 ", where 1 denotes –1. This adder receives three inputs, two redundant numbers {(X* – X**), (Y* – Y**)} and one redundant carry input (Cin*– Cin**) from previous stage, and produces two outputs {SUM (S* - S**), Cout (Cout* –Cout**)}. For the 1-bit new adder the intermediate carry input (Cin) is generated from the current addition, which is propagated only to the Cin- port of next higher bit position. Therefore, simultaneous parallel addition can be achieved.

When calculated, this adder gives only a 3-gate delay using the gate delay of one two-input only CMOS NAND gate. A gate level schematic of a Carry-Free Adder cell is given in Figure 76. This adder model is used as a basic two operands adder cell.

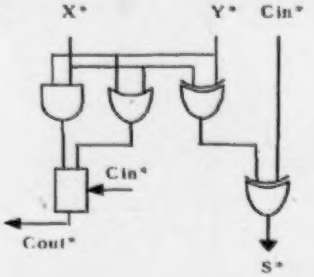


Figure 76 Carry-Free Adder Cell

5.1.6. Final Two Operands Carry-Free Adder

Fast operation is necessary in the final stage of the inner product processor to generate the final output. The two-operand adder is built using Carry-Free Adder cells as in Figure 76. Furthermore, while there are various possible implementations of a parallel adder, the carry-select (CS) adder is often used as optimum compromise between speed and power dissipation. Basically, a CS adder consists of several blocks, or stages, each executing two additions. One addition assumes that the input carry signal is '1', while the other one assumes it to be '0'. Thus, two sets of sum outputs are always pre-computed within each block. The final selection is performed once the actual input carry signal has been determined. The input carry signal also selects the output carry signal, which is used by the following block.

The optimal staging of a CS adder depends on supply voltage and process technology. However, the block sizes usually increase in size due to the multiplexing delay of the next carry. The staging from LSB to MSB is chosen to be 3-4-4-4. It must be pointed out that the Y inputs of this adder are the poly gates of transistors. Therefore, a full-swing signal must be applied to this input in order to avoid signal degradation. A Carry Select Adder with 3 and 4 bit Carry Free adders is used because of its high speed and regular layout. The block diagram below in Figure 77 illustrates the adder's configuration.

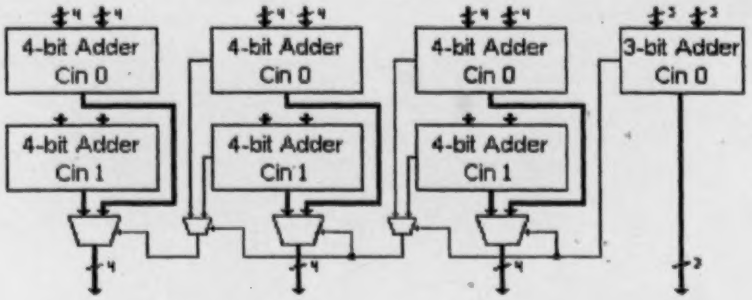


Figure 77 Final Carry Select Adder Block Diagram

5.1.7. Inner Product Processor Design Conclusion

The schematic of the entire six signed operands and 8-bit input resolution contains the 3-D Booth Encoder, Wallace Reduction Tree, and Carry–Select Adder as its main functional blocks. The output bit width is 16 bits. The entire design requires 61 4-2 compressors for the Fully Merged Arithmetic technique. If the Partially Merged technique had been applied instead the number of compressors would increase to 68 (210 more transistors).

To extend this design to 16-bit input operands would require some structural reorganization, mainly in the Wallace tree, and in the two-operand carry-select adders. More Booth Encoders, that can be instantiated by schematic duplication, would also be required.

Even though pipelining has not been used in this design, an analysis of where the circuit can be pipelined is still important since the throughput might have to be improved in the future. For 8-bit operands, the obtained speed of purely combinational computation is more than adequate for our application, since the considered system clock to be about 250 MHz to achieve an interactive rendering. Furthermore, pipelining is necessary if the 8-bit input operands are replaced with 16-bit representations. The architecture can be pipelined after the Booth Encoder, at each stage of the Wallace tree and before the input to the two operand adder, as shown in Figure 53.

Low-power optimizations are achieved at the logic level by employing FS and NFS CPL, and explained in details in following sections. Application of parallelism in the encoding stage also reduces power consumption.

5.1.8. Circuit Design Verification

Each circuit block was tested using Spectre simulations in the Cadence Analog Artist environment. The simulations were run using default parameters and initial conditions at 27°C. Test benches were created for the circuit blocks and the inputs stimulated using pulse generators. Furthermore, representative capacitive loads were placed at the outputs when testing each circuit block so that parametric timing analysis and power dissipation was more accurate. It is estimated that a single logic gate input, and the associated wiring, contributes around 6fF of load capacitance.

For each circuit, functional verification and delay and power dissipation analysis were performed. Delay analysis was measured for the worst case along the critical path between the 50% transition points of the input and output waveforms. Unless otherwise stated, power was measured as the average power dissipation for a worst-case output switching transition since CPL logic does not dissipate static power. The circuit nominal voltage was set to 1.8V.

The power dissipation was calculated using the Calculator tool as

$$P_{\text{average/switching}} = I_{\text{average}} * V_{dd} \qquad \textbf{Equation 58}$$

Although this method is not entirely accurate, it gives a rough estimate of the switching and quiescent power dissipation for each circuit block.

5.1.8.1. Booth Multiplier Design Verification

5.1.8.1.1. Booth Selector Verification

As explained above, the Booth Selector/Encoder uses groups of three bits to generate control signals for the Partial Product Generator so that it can perform the correct shifting and negation of the multiplicand. Since the Booth Selector control signals have to drive heavy loads (fanout of nine or ten for 'neg') the outputs of this circuit block were tested with 56fF capacitors. An exhaustive test was run for this circuit to ensure that the outputs corresponded with the Booth Encoding rules. The critical path through this circuit was determined to be from Yi-1 to NEG, as shown in Figure 68, and the measured delay is 331ps. The average power consumption of this circuit per transition was found to be 1.4mW.

5.1.8.1.2. Partial Product Generator Cell Design Verification

The Partial Product Generator of IPP was tested using four random test vectors. Since this circuit only needed to drive unit capacitance, the output was loaded with a 6fF capacitor. The circuit functions correctly with the worst-case delay along the critical path from Xi to PP of 242ps, as shown in Figure 69. The average power consumption of this circuit per transition was found to be 0.2mW.

5.1.8.1.3. Modified Booth Encoder Design Verification

The entire Modified Booth Encoder was tested for correct functionality in encoding two 8-bit values (one coordinate axes of inner product). The data input pattern of X = 10000000, Y = 00001111 was determined to cause the worst-case delay because of the need to generate the two's complement of the multiplicand and propagate the 2C addition across the partial products. The worst-case delay for generating all the partial products is 1.43ns. The average power consumption of this circuit per transition was found to be 3.45mW.

5.1.8.2. 15-bit Carry Select Adder Design Verification

The final two-operand Carry Select Adder was verified using the random test fixture. Since this is the final inner product output, it is assumed that the outputs will drive a unit capacitance of 6fF. Functional Testing was performed using several sets of random test vectors; two examples are given in Table 11:

Table 11 15-bit Adder Test Vectors

X	Y	S-result

000011110000111	111111110000000	000011100000111
111111111111111	000011110000111	000011110000110

Timing analysis was conducted, and the worst-case delay from input bit 0 to S<14>, since C<14> is ignored in the application, is 476ps. The power consumption of the circuit per transition (all outputs changing except for one) was found to be 3.98mW.

5.1.8.3. Inner Product Processor Design Verification

The entire Inner Product Processor circuit was verified. The stimulus is one set of input vectors since the simulation time for the entire circuit is 3.5 hours. The calculated intermediate results for each stage of encoding and column compression were verified according to the input vectors. The output vector agrees with the expected result. The delay time from initial input to final output for the worst-case bit in the circuit layout was measured to be 3.16ns (throughput = 316MHz).

Several power dissipation parameters were measured. For the entire processor, the average power for random vectors input at a frequency of 167MHz (6ns data rate) was 2.55mW. The average leakage power, when no inputs are being evaluated, is 20μW.

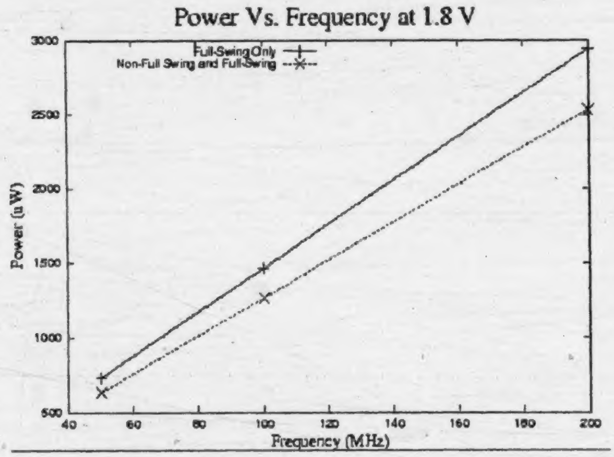
$$P_{av} = \frac{1}{T}\int_0^T p(t)dt \qquad \textbf{(Equation 59)}$$

The peak power dissipation at 167MHz was 29.4mW.

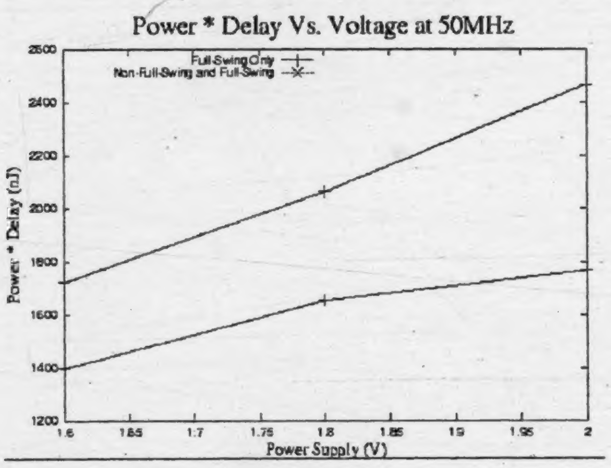
$$P_{peak} = i_{peak} V_{dd} \qquad \textbf{(Equation 60)}$$

The inner product evaluation time and power dissipation were simulated for voltages between 1.6V to 2.0V at frequencies from 50MHz to 200MHz. A circuit containing all full-swing internal nodes was compared against a circuit containing both reduced swing and full-swing internal nodes. Plots of the power dissipation and power delay product for different operating frequencies are shown in Figure 78 (a,b,c).

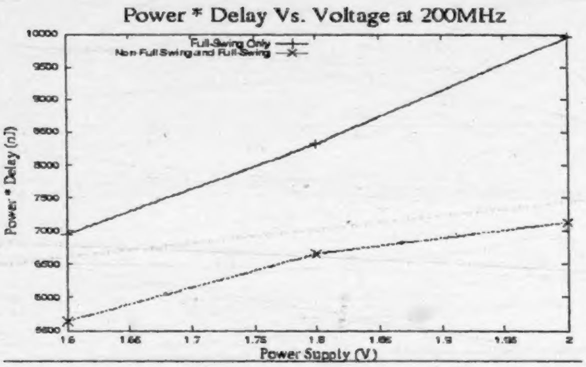
The average delay time at 1.8V (nominal supply voltage) is approximately 2.7ns for both implementations. Hence, the speed penalty for reduced swing internal nodes is very small. The power dissipation is reduced by 16% and the power delay product is reduced by 23-40% when reduced swing internal nodes are used. However, the noise margins are decreased as well.



(a)



(b)



(c)

Figure 78 (a,b,c) Inner Product processor performance results

5.1.9. Chip Fabrication and Evaluation

The inner product processor was fabricated by Taiwan Semiconductor Manufacturing Company (TSMC) on a test chip using a 0.18 μm, six-metal, single-poly CMOS process. The core layout was done by Julien Lamoureux. The IPP core layout is shown in Figure 79.

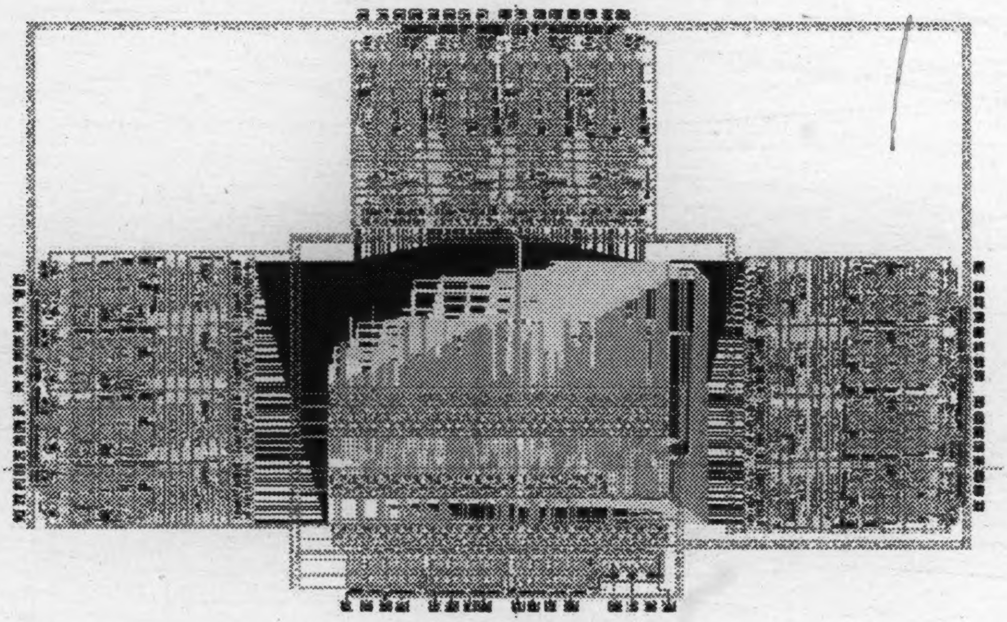


Figure 79 Inner product processor core layout

Custom manual routing was employed to minimize the interconnect lengths. The core area measures approximately 500 x 400 μm^2 and contains 9162 transistors.

5.1.9.1. Packaging

After fabrication, the silicon was packaged to enable its use at the board level. Several packaging options were available; however, two packages were finally considered:

- 84 pin-grid-array (PGA) package chosen because of its compatibility with the HP Tester, which could be used for functional testing of the chip.
- CFP80, which can perform at frequencies high enough to verify the inner product processor's parametric characteristics.

The second package was chosen. The manufactured IPP chip micrograph is given in Figure 80

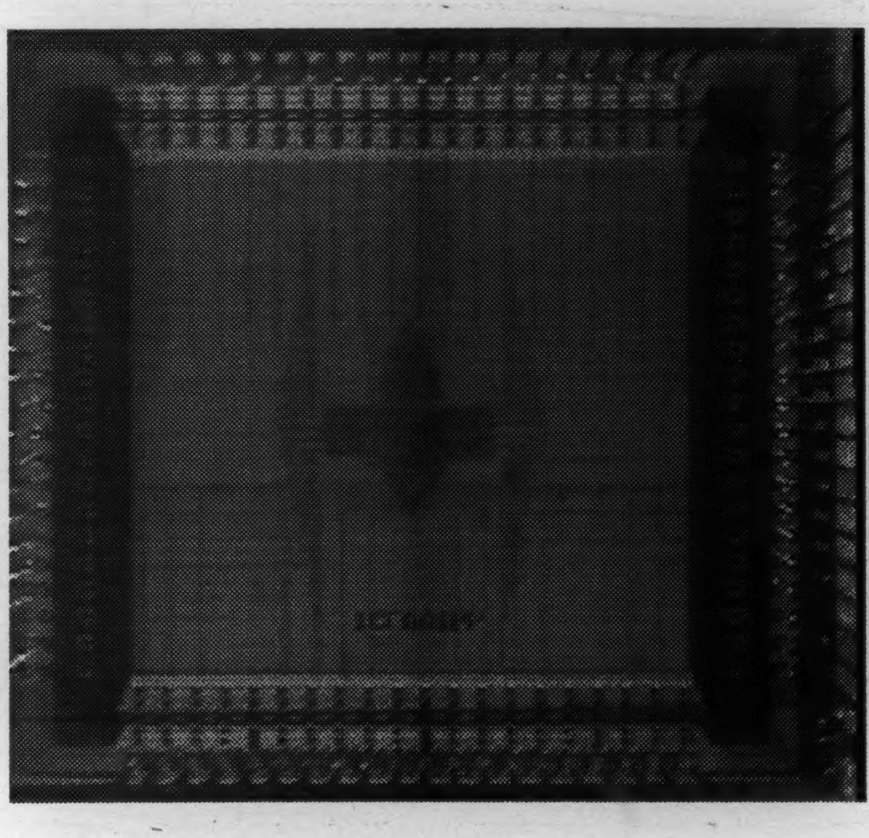


Figure 80 Inner product processor chip micrograph

The IPP's 16 power and ground pins are divided evenly into 8 power and 8 ground pins. These subsequent 8 pins are divided evenly between the core and pad ring for both power and ground. The pad ring requires a 3.3V power supply, whereas the core requires only 1.8V. To simplify the design and reduce fabrication cost, the IPP test fixture utilizes external power supplies, which can be connected using the test fixture's four power terminals. Because power-supply voltages change with advances in technology the test fixture's power supplies were left unspecified thus increasing the reusability of this design.

5.1.9.2. Chip Testing

5.1.9.2.1. Test System

The IPP chip was tested on a VXI Test System with a TH1000 Test Head.

5.1.9.2.2. Testing Results

Random test vectors were used to verify the IPP's functionality. The test showed:

- no errors were produced for the most positive (127) or most negative (-128) cases;
- errors occurred whenever the Bx, By, or Bz vectors were set to "00000000" and the even bits of the corresponding Ax, Ay, or Az vectors were set to 1.

Example:

Ax = "00000010" // 2nd bit is set

Bx = "00000000", or

Az = "10000000" // 8th bit is set

Bz = "00000000".

- Functional testing of the IPP has showed that the circuit is functional at frequencies of up to 20 MHz for all cases apart from cases where Bn = "00000000", and even bits of An = '1'. Six chips were tested and all performed in the same manner.
- Apart from the improper orientation of the silicon wafer within the PGA package, no fabrication errors were detected during the testing process.

Because the IPP is functional, it would be possible and worthwhile to perform parametric testing using a high speed, high pin count tester.

5.1.9.2.3. Error Detection

Because the error occurred for all three dimensions x, y, and z, it was somewhat obvious that the error stemmed from the Partial Product Generators because the PPG is the only block of the circuit where the x, y, and z components are processed separately. Performing more specific tests, shown in Table 12, helped to find the problem.

Table 12 Sample Errors (Dot_Product should equal 0)

Ax	00000010	00001000	00100000	10000000
Bx	00000000	00000000	00000000	00000000
Dot_ Product	11111111000000 00	11111100000000 00	11110000000000 00	11000000000000 00

In the PPGs, the A vector is Modified Booth encoded, which produces control signals for the internal PPGs. These control signals tell the internal PPGs to multiply B by 0, 1, or 2 and possibly to negate B. The problem occurs whenever B is '0' and A[n+2] is '1', which always corresponds to an even bit in A, because the MBE produces controls signals that inform the internal PPG negate B. When negating B=0, the 8 bits of B remain unchanged; however, the sign bit (bit 9) becomes '1' when it should remain '0'. The following example explains the error:

Ax = 00 000010 * Bx = 00000000

Modified Booth Encoding of Ax produces four factors

Ax[7-5] = "000" -> 0

Ax[5-3] = "000" -> 0

Ax[3-1] = "000" -> 0

Ax[1-0]+'0' = "100" -> (-0)

Adding all the partial product we get

Bx * (-0) -> (1111111)10000000 // error occurs here at the ninth bit

```
Bx * 0          -> (00000)000000000
Bx * 0          -> (000)000000000
Bx * 0          -> (0)000000000
-----------------------------------------------------
                    111111100000000
```

The calculation should clearly have produced “000000000000000” because zero multiplied by anything is also zero; however, “111111100000000” was produced.

5.1.9.2.4. Error Correction

The error detected during functional testing was analyzed, and it found to occur in sign extension block, where the combination all “0”s should be designed as a special case, and generally excluded from computations. The correction has been done by an additional circuit “glue” that asserted a parallel set of the FS NXOR gates and a MUX, which outputs the final result without calculations in case of all “0”s.

5.1.10. Circuit VHDL Implementations

Each sub-component has “soft” (VHDL) and “hard” (ASIC) implementations. Thus, the design might be easily transferred to another technology or implemented into FPGA.

These are the basic circuits that create the package components for SWF system. For the FPGA implementation the VHDL library is located in the swf_sys_pkg file. The following components are included: Booth encoder, partial products generator, sign extension simplification block, Wallace tree stages, 4-2 compressor, carry free adder, carry save carry free adder, counters, pipelining FF, re-sampling and synchronizing FF (SYNC FF), comparator, half – adder, full adder, multiplexer, de-multiplexer, etc. The basic components, such as logic gates, are composed into package file work_lib_pkg. All the given components are described in VHDL and tested.

To synthesize the code (using Synopsis) the sub-components used the FS and NFS CPL library mapped to the hierarchically higher components providing the highest performance. The design and verification of the FS and NFS CPL cells is provided in the following section.

5.2. Logic Design

5.2.1. Low Power Logic Design

The use of a Modified Booth/Wallace Tree multiplier and Merged Arithmetic algorithm results in reduced hardware and associated power savings. Other low-power techniques will be accomplished at the logic level, where new structures of transistor configurations are used to reduce power consumption while maintaining or improving the performance. Furthermore, the fanout load of every gate output is taken into account and large transistor buffers are inserted where necessary to decrease rise and fall times.

In terms of logic function implementations, pass-transistor logic has emerged as an attractive replacement for conventional static CMOS. Fewer transistors are required in pass-transistor switch networks, which result in lower input gate capacitance, faster switching speeds and reduced power consumption. All of the arithmetic units and encoders were implemented using pass-transistor logic styles. Sub-components circuits were designed and then fabricated in 0.18 μm technology. The applied nominal operating voltage was initially set at the recommended 1.8V, but further eventually might be reduced depending on whether the timing constraints can still be met.

5.2.2. Non-Full Swing Pass Transistor Logic (NFS CPL) Cells

Complementary pass-transistor logic (CPL) has been claimed to be a power-efficient logic style. The advantages of CPL style include:

- Complex circuits can be realized with small number of transistors
- Circuits are differential
- Improved noise resilience
- Due to reduced threshold, below $|V_{tr}|$, switching speed is increased

Forming compact functional circuits with gates that have reduced output swing allows for significant decreases in power consumption. However, a major problem of CPL lies in the weak driving ability of the outputs. In the classical CPL implementations, output buffering compensates for this. NFS gates (without level restoration devices) could be used when the generated outputs are not applied to the polysilicon gate of another transistor. When the signal has to be applied to a transistor gate, level restoration is accomplished through cross–coupled PMOS pull-up devices.

All of the basic cells for the SWF sub-components are designed using the NFS-CPL logic style [17]. Full-swing and non-full-swing versions of all the basic cells were built. All NFS outputs must drive those inputs that accept NFS logic HIGH, while FS node can be used to drive any input. The connections of the output nodes to input nodes define whether FS or NFS cells are used. If the signal is applied to the transistor's gate, then the FS cell must be used. If the signal is applied to a transistor's source/drain, the NFS cell might be utilized. Any long connection of pass-transistors is undesirable and, therefore must be kept to a minimum. To insert NFS nodes for low power operation and to maintain speed performance, the longest serial connection is limited to three transistors before a polysilicon gate terminates it.

5.2.2.1. Process and Device Parameter

The design was implemented in a 0.18μm n-well CMOS process offered by TSMC. Unless otherwise specified, all NMOS pass transistors were made minimum size since, to a first order, increasing the size of the devices in pass transistor logic has no net impact on the propagation delay unless the load capacitance is a dominating factor. Increasing the (W/L) ratio reduces the resistance, while simultaneously increases the diffusion capacitance. However, some benefit can be gained for long chains of pass transistors by progressive sizing. This is because, the first device in the chain, farthest from the output, has to charge/discharge the parasitic capacitances of all other devices in the chain. This device should be made larger than the second one, and so forth. However, since the circuits were put together using a cell-based approach, transistors closer to the output are not made smaller than those farther from the output. While this design decision may reduce the speed of the circuit, great care was taken to ensure that a series connection of pass transistors never exceeded 3; thus alleviating the problem to a large extent.

The PMOS cross-coupled level restoration devices were made minimum size. This is because the NMOS network contains minimum size devices and the use of large PMOS transistors will prevent the high-level charge stored in the output latch from discharging.

The use of minimum size PMOS transistors slows down the transition of the high-level to V_{dd} although the performance penalty is minimal.

The optimum transistor size is one of the important parameters because it affects the transistor performance. Figure 81 shows the delay and power characteristics of the NFS CPL AND/NAND gate as a function of PMOS transistors in pull up circuit and those in the circuit network.

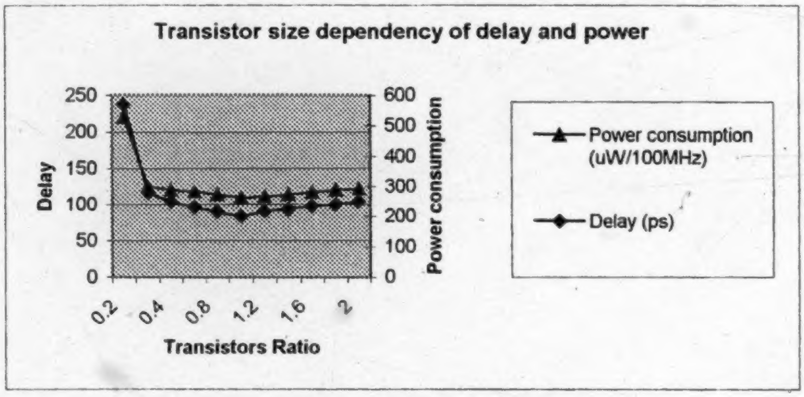


Figure 81 Transistors size dependence

Simulations have been done using 0.18 um CMOS technology. The results show that the optimal ratio between PMOS and NMOS transistors is 1. Static CMOS inverters were inserted to drive heavy loads. In these cases, the NMOS transistor is made rather large and the PMOS transistor is made twice as large to account for reduced holes' mobility in PMOS devices and to maintain symmetrical voltage transfer characteristics. It attempts to exploit the non-full-swing nature of NMOS pass-transistor logic instead of restoring all threshold voltage drops to full swing.

5.2.3. Basic Gate Design

A library of full swing (FS) and non-full swing (NFS) logic gates was developed and verified. Two examples of this library are shown in Figure 82.

The basic rules concerning this pass-transistor logic style are threefold. First, and foremost, the output of a non-full swing logic gate must connect to the input of another logic gate (NFS or FS) formed by the drain/source of a transistor.

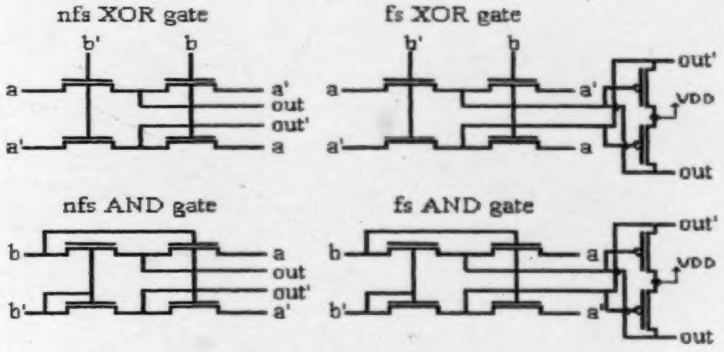


Figure 82 NFS and FS logic cells

This is vital for the purposes of minimizing delay and maximizing noise margins because connecting a NFS output to the gate of another transistor would result in a threshold voltage drop of $V_{dd} - 2V_{th}$.

The testing results of basic cells have shown that although cells implemented with NFS nodes are slower than those containing all FS nodes (16% decrease in speed), the power–delay product is improved by more than 50%.

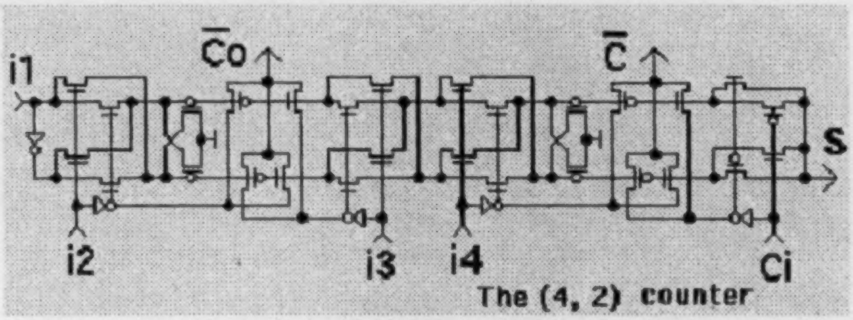
5.2.3.1. 4-2 Compressors

The 4-2 compressor is the most important basic cell in the dot product processor, and, respectively, in the matrix multiplication processor. Below in Table 13 several designed basic cells of different logic styles are reviewed and analyzed.

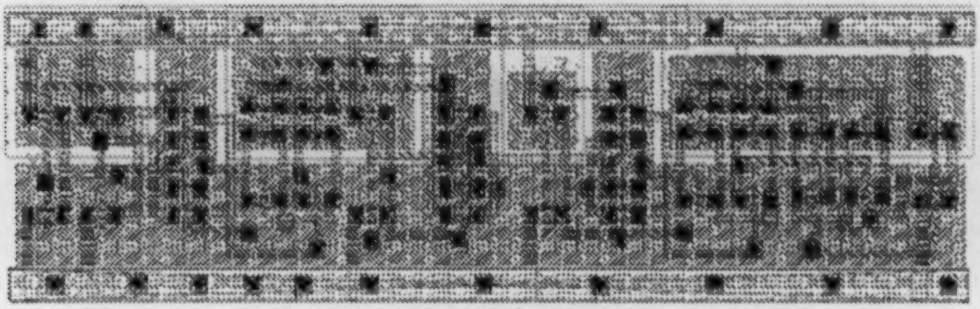
Table 13 4-2 compressors review

Source	Logic style	Power cons.MW/ 200MHz	Area Sq.um	Metal used/ technology	Delay ns	Trans. count
[34]	NFS CPL	0.1431	10x12	metal 4 0.18um	0.279	30
Std. cell				0.25um	0.680	
[33]	TG	0.1830	11x41	metal 2 0.25um 0.18um	0.622 0.407	38

Table 13 gives the review of 4-2 compressors found in literature, standard cells, and one of the tested designs, which was considered as an alternative 4-2 compressor for the SWF sub-components design. Figure 83 shows the 4-2 compressor, given in Table 13 and designed by Lin [33], which is based on transmission gates (TG) logic.



(a) TG based schematic



(b) Custom layout

Figure 83 (a,b) 4-2 compressor TG based

Layout has been made in 0.25 um CMOS technology. The design was tested, and the performance results are given in Table 13, and in [33].

The same design has been transferred to other two technologies: 0.18um and 0.13um, and simulated. Performance results are concluded respectively in Figure 84 and Figure 85.

A TG based 4-2 compressor was considered to be used in the Wallace Tree design. However, the high performance NFS 4-2 compressor, described below, has been chosen.

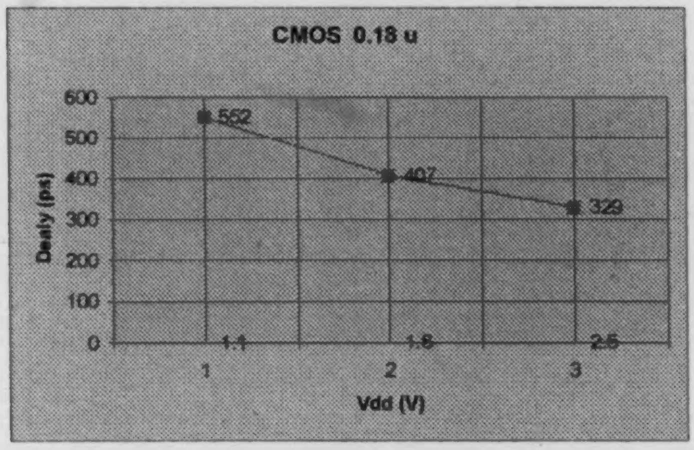


Figure 84 TG based 4-2 compressor performance in 0.18 um CMOS

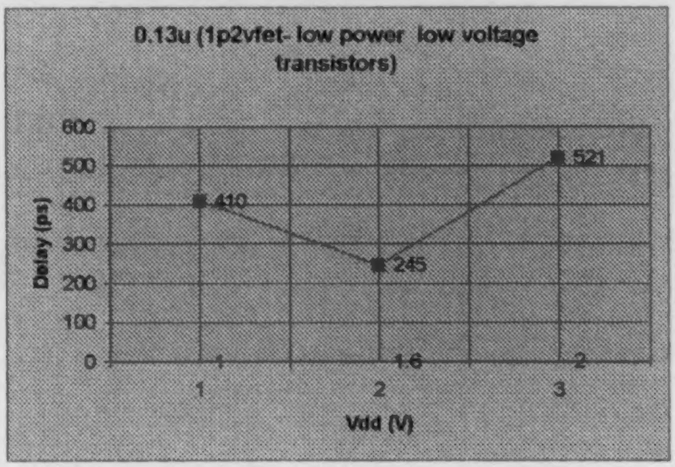


Figure 85 TG based 4-2 compressor performance in 0.13 um CMOS

5.2.3.1.1. 4-2 Compressor Design

As was shown in the development of a 4-2 compressor, the speed of the arithmetic block is only slightly reduced by the reduced swing of internal and external nodes when special care is taken when routing the multiplexers, which form the compressor. Furthermore, special considerations have to be taken into account when using the compressor to form a tree adder. For CPL multiplexers, the nodes that must accept full-swing signals are the select inputs, while for the other basic two-input gates an arbitrary selection of the 'b' input in Figure 82, must be full swing. Next, if the outputs need to drive a signal line with greater than unit fanout, then a static CMOS inverter, or chain of inverters, have to be placed at the outputs since pass logic has no active drive. In order to avoid static power dissipation for these inverters, the input to the inverter must be full swing. This is because the reduced high level across an NMOS pass transistor will not turn the inverter PMOS fully off, resulting in crowbar current. Last, one must be careful of long serial connections of pass transistors in CPL logic.

Due to the speed degradation this is even more crucial when many of the nodes in CPL are non-full-swing. The 4-2 compressors are designed so that connecting these cells in a Wallace Tree will at most result in a series connection of three pass-transistors before a polysilicon transistor gate terminates the signal. In other NFS-CPL gates, limiting the serial connections to three or four has been found to be optimal.

Each 4-2 compressor receives five inputs: In0, In1 In2, In3 and C_{in}, compresses them, and generates three output bits: C_{out}, Sum and Carry. One of the output carry signals C_{out} is generated based on In0, In1 and In2 only, and the input C_{in} and the sum of the above term is used to generate Sum and Carry. Thus, there is no "horizontal" propagation of carry signals across a Wallace tree. A schematic of the NFS 4-2 compressor is shown in Figure 86.

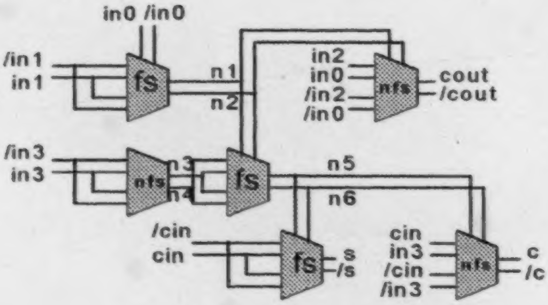


Figure 86 NFS 4-2 compressor

The implemented compressor consists of two types of CPL pass–transistor multiplexers: Non-Full-Swing (NFS) and Full Swing (FS), where the level restoration is provided by cross–coupled PMOS device. The schematics of full swing and non-full swing pass-transistor multiplexers, and its incorporation into a 4-2 compressor [17] are shown in Figure 87.

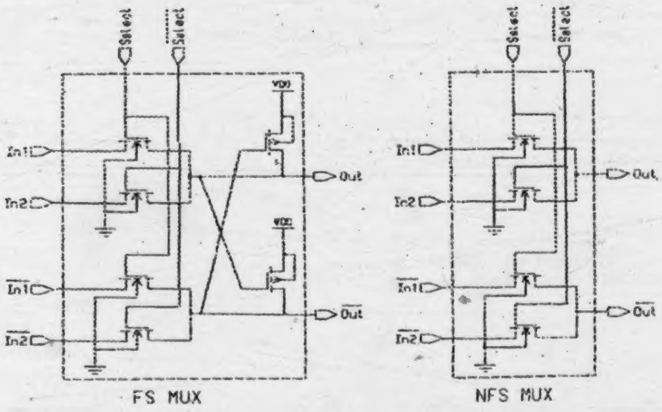


Figure 87 Full-Swing and Non-Full-Swing Multiplexers

In each compressor, Figure 86, only the internal nodes n1, n2, n5 and n6 and the output Sum are pulled up to V_{dd} for a logic HIGH, while the rest only reaches Vdd-V_{tn} where V_{tn} is the threshold voltage of the n-channel MOSFET modified by the body effect. NFS nodes are C (carry), C_{out}, n3 and n4. Again special care must be taken to connect NFS nodes to inputs that can accept NFS. Thus, the output node C_{out} must be connected to NFS acceptable inputs in the same level of the Wallace Tree. These are represented by C_{in}, In1 or In3. The Carry signal propagates to the next level of compressors and must also be connected again to NFS input nodes. The Sum output, on the other hand, being a FS node, can be used to drive any of the next stage compressor inputs.

With this technique, approximately 50% of the nodes within the Wallace tree are NFS. Furthermore, since only two of the 4-2 compressor inputs require FS, only half of all partial products are required to achieve full swing. This leads to significant power reduction in the processor.

Finally, as compared to a 4-2 compressor implemented using Static CMOS, the NFS-CPL compressor requires 30 transistors while the CMOS one requires 68. Thus, the circuit area reduction is significant.

Because, only 50% of the nodes in a Wallace Tree Adder, built using NFS-CPL, is full swing (FS), less energy is required for the pull-up operation and hence less far less power is dissipated.

5.2.3.1.2. 4-2 Compressor Design Verification

The CPL 4-2 Compressor was tested using random vectors. Since each output only drives unit capacitance, the outputs were fitted with a 6fF load. The outputs were verified for the following input and output vectors:

Table 14 4-2 Compressor Test Vectors

I3	I2	I1	I0	C_{in}	C_{out}	S	C
0	**0**	**1**	**1**	**1**	**1**	**1**	**1**
0	**0**	**0**	**1**	**1**	**0**	**0**	**1**
0	**0**	**1**	**0**	**0**	**0**	**1**	**0**

Figure 88 shows the CPL NFS 4-2 compressor functional timing Spectre 0.18 um CMOS simulations.

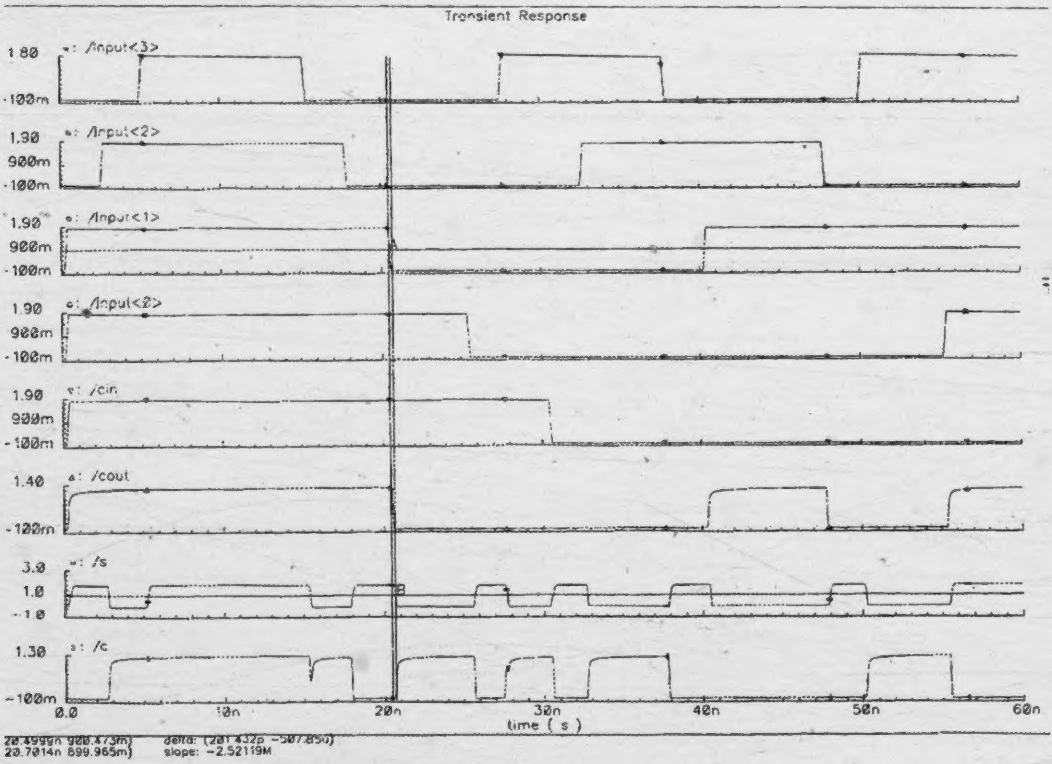


Figure 88 4-2 compressor functional timing diagram

Furthermore, hand analysis was used to verify this functional block using the truth table. The worst-case delay from I1 to S was found to be 201ps. The power consumption of this circuit per transition was found to be 0.2mW. Since this functional block is so vital to the correct operation of the entire circuit, several sets of measurements were done for a 4-2 compressor built as from the fastest available standard and from described in the previous chapter as well.

The worst-case delay for the standard cell circuit is measured to be 468ps, which is much slower than the NFS-CPL circuit. The power consumption of the standard cell circuit per transition was found to be 0.5mW.

5.2.3.1.3. Flip Flop

The power consumption of the clocking system, including the clock driver and storage elements, may occupy a large portion of the total power in a pipelined multiplier (45 – 70% of the total power). As an extension to this finding, the design of the storage elements not only determines the power consumption of the storage elements themselves, but also influences the power consumption of the clock driver. Therefore, the design of the storage element is the key to a low power multiplier-based circuit. Since the storage in each pipeline stage only needs to be temporary for the purposes of computation, dynamic storage elements should be used. Furthermore, it is desirable that these storage elements operate with true single phase clocks (TSPC). Many low-power and high-speed dynamic CMOS latches and flip-flops have been proposed in literature [24]. Among them the pulse-triggered true single-phase clocking FF (PTFF) [23], shown in Figure 89, features a simple structure with only one clocked transistor per storage element.

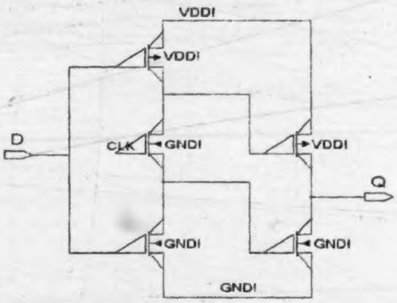


Figure 89 PTFF schematic

Thus, the power consumption of both the storage elements and the clock driver can be reduced significantly. The PTFF can operate as an edge-triggered element through applying narrow triangular wave pulse trains. The output is charged or discharged at the rising edge of the clock, and the input data appears at the output after the clock goes low. To achieve minimum power dissipation, each transistor should be minimum size to reduce the source/drain capacitance contribution. For the PTFF, it is shown that this flip-flop saves up to 47% of the power compared to other flip-flop and latch designs while maintaining equivalent performance.

Regular pipelining registers are built on the PTFF [23]. Table 15 gives the comparison of the chosen PTFF with technologies standard cells.

Table 15 Library dependent FF parameters

Technology	Set up time of FF	Tau parameter of FF	FF delay
Toshiba 0.18um	0.350 ns	0.151ns	0.420 ns
TSMC 0.18 um	0.390 ns	0.178 ns	0.255 ns
TSMC 0.13um	0.186	0.083 ns	0.275 ns

The operation of this special flip-flop requires the generation of a narrow-pulsed train. A simple circuit to generate these pulses from a nomimal 50% duty cycle clock can be found as shown in Figure 90.

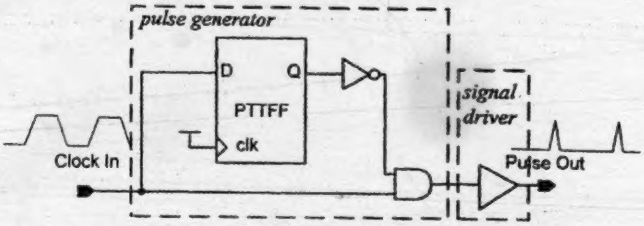


Figure 90 Pulsed Clock Generator

The rise/fall time of the clock signal also has an effect on the power dissipation of the storage elements. The power dissipation was found to increase more than linearly with a linear decrease in the rise/fall times. Thus, the rise/fall time of the clock should be adjusted for this performance/power dissipation trade-off.

5.2.3.1.4. PTFF Design Verification

Simulations were conducted for the most promising latch/flip-flop, the PTFF. It is assumed that the flip-flop, which forms each bit of the pipeline register, will drive a unit capacitance of 6fF. The timing diagram is given in Figure 91.

The clock spike is 1ps in duration with a rise and fall time of 100ps. The delay required for latching an input is 175ps. Power dissipation for latching a '1' and a '0' were determined to be 392μW and 336μW respectively.

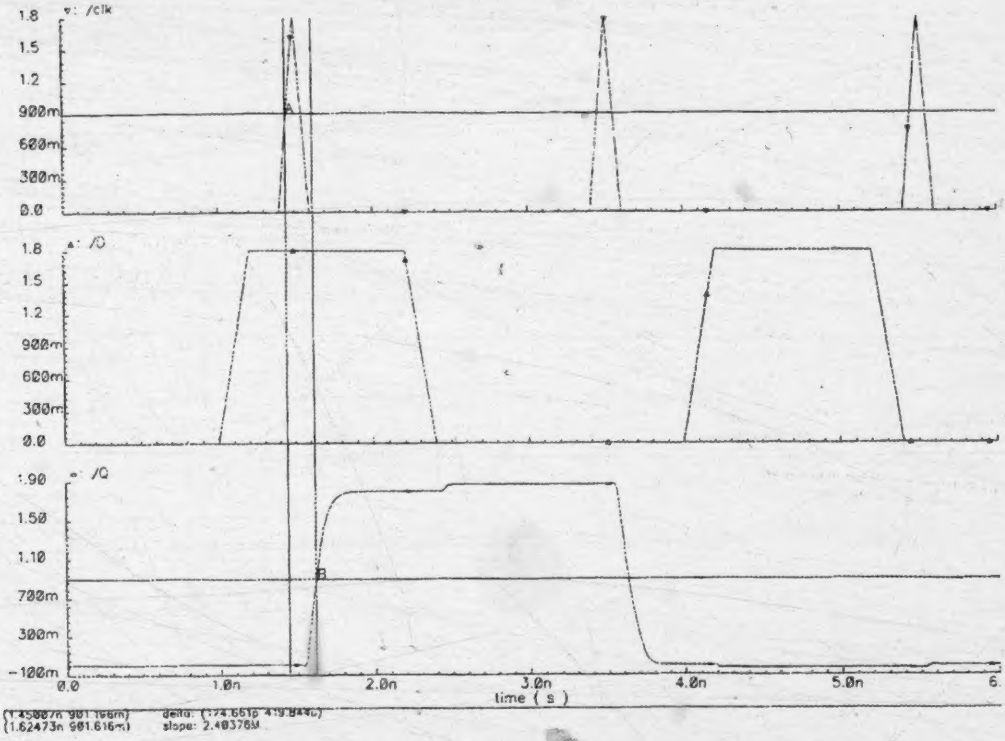


Figure 91 PTFF functional timing diagram

5.2.4. Chapter Summary

In this chapter we described the design of the circuit level sub-components: inner product processor, widely used for matrix multiplication, gradient computation and angle cosine definition, Booth encoder, Booth multiplier, and Carry-Select Carry-Free adder for various bit configurations. Sub-components have been designed, verified, manufactured and tested.

The high performance and low power NFS transistor logic, used for the sub-components design, has been designed and tested. This logic has some specific application aspects since the great attention must be paid how NFS gates are connected to avoid signal degradation problem. However, this logic style significantly decreases power consumption, while keeps all advantages of CPL.

The next chapter is a conclusions chapter, divided as a final summary and a future work. Those sections conclude the work done, and the work to be done.

Chapter 6

6. Conclusions and Future Work

6.1. Final Summary

Real-time volume rendering has become essential during the last decade; however, very few rendering systems are available to provide an interactive rendering.

Real time rendering means the smooth rendering of the volume with processing speed 30 frames per second. To render 1024 x1024x1024 voxels volume within the real time rates each voxel must be processed for not less then 31.4 ns, and for 512x512x512 voxels volume voxel processing time should not exceed 125.8 ns.

Thus, it becomes obvious that to accomplish these rates, general - purpose computers are not sufficient enough. However, on the other hand, multiprocessors with sufficient computational power are too expensive. Therefore, a rendering system built as Plug and Play card is the optimal solution.

This thesis described a new volume rendering system employing the Shear-Warp Factorization volume rendering algorithm. We explained why this algorithm has been chosen for the implementation, its advantages and disadvantages.

The Shear-warp factorization algorithm is the fastest among the existing direct volume rendering algorithms. That factor was the decisive for our choice, since our target is to achieve a real-time volume rendering rate. The SWF algorithm combines the advantages of two classes of algorithms: image –order ray casting and object-order splatting. Also, employment of the bi-linear interpolation significantly simplified calculations, and run-length encoding of the classified volume reduced the number of computation by skipping the transparent voxels. Therefore, the SWF algorithm is the optimal choice for the implementation.

None of the existing "hardware" rendering systems built on the ray casting class of algorithm computes perspective projections, since it requires additional computations. Our implementation supports parallel and perspective projections; complete mathematical derivations for rendering a volume with either perspective or parallel projections fully explain the design computational steps and structure.

The main feature of the SWF algorithm is that there is no single performance bottleneck remaining. The rendering time is divided roughly equally between essential computations, re-sampling, shading, gradient computations and compositing, overhead due to the coherence data structure and control logic, and memory overhead. The time slicing architectural approach takes the advantage of this feature. Time sliced pipelined architecture and components reusability allows simultaneous processing of several tasks. Moreover, this approach provides integration and continuous communication of "hardware " components with "software" components, and interactive user interface.

The designed system has four hierarchical levels: system level, components level, circuits level, and logic level. Speed and power optimizations have been done on each of them. The SWF volume rendering system has been designed with low power aspects. Main power consumption reductions were achieved on the logic level by employing Non-Full Swing pass transistors logic, and on the components and system level by implementing pipelining and parallelism; speed optimizations were mainly done on the circuit and system level.

The component level contains re-usable and re-configurable blocks, described in the volume rendering pipeline. These components are combined in SWF system through the Context RAM and the External Communication Interface. Components are built from the basic sub-components, or circuits, which, in turn, are built from the logic.

The circuit level contains adders, encoders, inner product processor, and other blocks, which form the higher level components such as Matrix Multiplication Processor, Gradient Processor, Composer and others. Logic and circuit levels sub-components create the custom sub-components library, which is mapped to the components level. All designs were exhaustively verified.

The Shearer- Composer was implemented in an FPGA; it is able to process not only parallel projections, but perspective projections as well. Its functionality verified, and the results were published. The Inner Product Processor was fabricated and tested, and the results were published as well.

The achieved results allow us to conclude that SWF VRS is capable of rendering the 512^3 voxels volume within real – time rates.

6.2. Future Work

As was mentioned in the Chapter 4, the Shading Processor VLSI implementation is a part of the future work. After the simplifications were made to the Shader's mathematical model, shown in the Chapter 4, we face now quite simple implementation, which does not require complex sub-components and time and power consuming computations.

Classifier software implementation, which incorporates the algorithm correction explained in the Chapter 4 is a part of the future work as well.

Therefore, the only two designs are left to complete the components' hierarchical stage.

Written VHDL code might be programmed into FPGA, or synthesized in Synopsis.

6.2.1. Future Directions for the VRS Performance Improvements

Several proposals for increasing processing speed were found and analyzed. A proposal by Kanus in [56] binary masking is compensated by run length encoding used in the shear-warp factorization, which eventually fulfills the same task: skipping transparent voxels. Data set partition proposed in [2] has significant preprocessing time. The authors do not consider time devoted to the data partition as a computational time, although in reality it will be included in computational time in an interactive system.

One of the most interesting ideas is to apply the coding LZ77 algorithm, based on the data redundancy for computations. Employment of this method will immensely speed up the computations. However, application of this compression algorithm is effective only when we deal with regular and redundant data structures: than the amount of computations will be reduced in the linear dependence on the amount of redundant structures. Therefore, as a preprocessing step we have to define the volume structure, and whether the LZ77 compression employing is effective.

References

[1] Drebin, R.A., Carpenter, L. and Hanrahan, P. Volume Rendering. *Computer Graphics*, 22, pp.65-74, August, 1988.

[2] Kentaro Sano, K., Kitajima, H., Kobayashi, H. and Nakamura, T., Parallel Processing of the Shear-Warp Factorization with the Binary-Swap Method on a Distributed-Memory Multiprocessor System. *In: PRS 1997. Proceedings of the IEEE symposium on Parallel rendering*, pages 87-ff.

[3] Lacroute, P., Levoy, M. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. In *Computer Graphics, Proceedings, Annual Conference Series (SIGGRAPH '94)*, Orlando, pp. 451-459, 1994.

[4] Lacroute, P. Volpack: *A Volume Rendering Library*, Stanford University, 1995.

[5] Philippe Lacroute, Ph.D. dissertation, *Technical Report CSL-TR-95-678*, Stanford University, 1995.

[6] Levoy, M. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3): 245-61, July 1990.

[7] Margala, M., Durdle, N., Juskiw, S., Raso, J. V., Hill, D., A 33 MHz 16-bit Gradient Calculator for Real- Time Volume Imaging. *Comput. & Graphics*, Vol. 19, No. 5, pp. 679-684, 1995

[8] Neumann, Ulrich. Interactive Volume Rendering on a Multicomputer. *not known*, 1992.

[9] Patterson, D. and Hennesy, J., Computer Architecture: A quantitative approach, Morgan Kaufmann, 1991.

[10] Yen, S.Y., Rubin, G.D., and Napel, S. Fast Sliding Thin Slab Volume Visualization. In *Proceedings of the 1996 Symposium on Volume Visualization*, pp.79-86, San Francisco, October 1996

[11] Foley, J., van Dam, A., *Computer Graphics, Principles and Practice*, Addison-Wesley Publishing Company, Inc. Second Edition. 1987.

[12] Lichtenbelt, B., Crane, R., Naqvi, S. *Volume Rendering, Hewlett-Packard Professional Books*

[13] Swartzlander, E. E., “Merged Arithmetic”, *IEEE Transacitons on Computers*, c-29 (10): 946-950, October 1980

[14] Feiste, K. A. and Swartzlander, E. E. Merged Arithmetic Revisited, *1997 IEEE Workshop on Signal Processing Systems. SiPS 97 Design and Implementation.* Formerly *VLSI Signal Processing*, 1997, pp. 212-221.

[15] Bellaouar Abdellatif; *Low-Power Digital VLSI Design Circuits and Systems*, Kluwer, 1995.

[16] Wallace, C.S., A suggestion for a Fast Mulitplier, *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14-17, Feb. 1964

[17] Law, C.F.; Rofail, S.S.; Yeo, K.S., Low-power circuit implementation for partial-product addition using pass transistor logic, *IEE Proceedings-Circuits, Devices and Systems*, vol. 146, no. 3 p. 124-9, June 1999

[18] Song Paul J. and De Micheli Giovanni, Circuit and Architecture Trade-offs for High Speed Mulitplication, *IEEE Journal of Solid-State Circuits*, vol. 26, no.9, September 1991

[19] Choo Iijoo; *et al*, A Novel Fast Parallel Signed-Digit Hybrid Multiplication Scheme for Digital Systems, *IEEE Canadian Conference on Electrical and Computer Engineering*, p. 630-35, May 2000

[20] Hsiao, S.F.; *et al*, Design of high-speed low-power 3-2 counter and 4-2 compressor for fast multipliers, *Electronic Letters*, vol.34, no.4, February 1998

[21] Yano K.; *et al*, "A 3.8-ns CMOS 16x16-b multiplier using complementary pass-transistor logic," *Proc. IEEE CICC'89*, 10.4.1, May 1989

[22] Rabaey Jan M., *Digital Integrated Circuits*, Prentice Hall, 1996

[23] Wang Jinn-Shyan; *et al*, Design of a 3-V 300 MHz Low-Power 8-b x 8-b Pipelined Multiplier Using Pulse-Triggered TSPC Flip Flops, *IEEE Journal of Solid State Circuits*, vol. 35, no.4, p. 583-92, April 2000

[24] Yuan Jiren; *et al*, New Single-Clock CMOS Latches and Flip Flops with Improved Speed and Power Savings, *IEEE Journal of Solid-State Circuits*, vol.32, no.1, p.62-9, January 1997

[25] Law C.F.; *et al*, A Low-Power 16 x 16-b Parallel Multiplier Utilizing Pass-Transistor Logic, *IEEE Journal of Solid-State Circuits*, vol. 34, no .10, October 1999

[26] Liu, D., and McCluskey, E.J., Design of CMOS VLSI Circuits for Testability, *CICC'86*, pp. 421-424; *Journ. of Semicustom IC's*, Vol. 4, No. 4, pp. 5-10, June 1987.

[27] McCluskey, E.J., VLSI Design for Testability, *1984 Symposium on VLSI Technology*, San Diego, CA, pp. 2-5, Sep. 10-12, 1984; *DASC*, pp. 523-530

[28] Touba, N.A., and McCluskey, E.J., Automated Logic Synthesis of Random Pattern Testable Circuits, *Proc. 1994 Int. Test Conf.*, Washington, D.C., pp. 174-183, Oct. 2-6, 1994.

[29] Touba, N.A., and McCluskey, E.J., Transformed Pseudo-Random Patterns for BIST, *13th IEEE VLSI Test Symp.*, pp. 410-416, Princeton, NJ, Apr. 30 - May 3, 1995.

[30] Touba, N.A., and McCluskey, E.J., Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST, *Proc. 1995 Int. Test Conf.*, pp. 674-682, Washington, D.C., Oct. 23-25, 1995.

[31] Armstrong, J.R., Gray, F.G., VHDL Representation and Synthesis, Prentice Hall PTR, 2000.

[32] Calhoun, P.S., Kuszyk, B.S., Heat, D.G., Carley, J.C. and Fishman, E.K., Three-dimensional Volume Rendering of Spiral CT Data: Theory and Method, *Radiographics*. 1999; vol.19, pp.745-764

[33] Lin, R., Margala, M., and Kazakova, N. Novel Self-Repairable Parallel Multiplier Architecture, Design and Test., *Proceedings of IEEE PM ASIC Symposium*, Taiwan, August 2002

[34] Kazakova, N., Sung, R., Durdle, N. G., Margala, M., and Lamoureux, J., Fast and Low-Power Inner Product Processor, *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2001),* Sydney, Australia, May 2001.

[35] Kazakova, N., Margala, M. and Durdle, N. G. A VHDL Implementation of a Shearing Unit for Shear-Warp Factorization Volume Rendering, in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering 2000*, Halifax, pp.1118- 1122.

[36] Kim.K. and Wittenbrink C.M., Extended Specifications and Test Data Sets for Data Level Comparisons of Direct Volume Rendering Algorithms, *IEEE Transactions on Vizualization and Computer Graphict,* vol.7, NO.4, October-December 2001, p.299.

[37] Kanopoulos N., Yasanthavada N., and Baker R., Design of an Image Edge Detection Filter Using the Sobel Operator, *IEEE Journal of Solid State Circuits,* vol. 23, NO. 2, April 1988, pp.358-367.

[38] Bright, S. and Laflin, S., Shading of Solid Voxels Model, *Computer Graphics Forum,* 5(2), pp.131-137, June 1986

[39] Juskiw S., Technical Notes 1-4, May1992.

[40] Snip, K. A., Elliott, D. G., Margala, M. and Durdle, N. G., Using Computational RAM for Volume Rendering, in Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, Washington, D.C., September 2000, pp. 253-257.

[41] Snip, A., Elliott, D. G., Margala, M., and Durdle, N. G., Investigation into the Use of Processors for Volume Rendering, in Digest of Papers, Memory Wall Workshop, International Symposium on Computer Architecture (ISCA), 10 pages, June 2000.

[42] Margala, M. and Durdle, N.G., A Gradient Processor for High Speed Medical Imaging, in Proceedings of the 7th Annual IEEE International ASIC Conference and Exhibit, pp.246-249, 1994.

[43] Margala, M., Durdle, N.G., Juskiw, S., Raso, V.J. and Hill, D., A 33MHz 16-bit Gradient Calculator for Real-Time Volume Imaging, in Proceedings of the 9th Eurographics Workshop on Computer Graphics Hardware, pp.80-85, 1994.

[44] Pfister H., Jan H., Knittel J. y Lauer H. and Seiler L., The VolumePro Real-Time Ray-Casting System, Mitsubishi Electric Corp. Report, 2001

[45] Margala, M. and Durdle, N.G., Low-Voltage Power-Efficient BiDPL Adder for VLSI Applications, Microelectronics Journal, vol.30, no.2, pp.193-197, February 1999.

[46] Margala, M. and Durdle, N.G., Non-complementary BiCMOS and CMOS Logic for Low-Voltage Low-Power Operation - A Comparative Study, IEEE Journal of Solid-State Circuits, vol.33, no.10, pp.1580-1585, October 1998.

[47] Margala, M. and Durdle, N.G., Low-Power 4-2 Compressor Circuits, International Journal of Electronics, vol.85, no.2, pp.165-176, August 1998.

[48] Margala, M. and Durdle, N.G., 1.2V Full-Swing BiDPL Logic Gate, Microelectronics Journal, vol.29, no.7, pp.421-429, July 1998.

[49] Margala, M. and Durdle, N.G., Novel Low-Voltage Low-Power Full-Swing BiNMOS Logic Gate, International Journal of Electronics, vol.84, no.5, pp.487-498, May 1998.

[50] Law C.F.; *et al*, A Low-Power 16 x 16-b Parallel Multiplier Utilizing Pass-Transistor Logic, *IEEE Journal of Solid-State Circuits*, vol. 34, no .10, October 1999

[51] Shams A.M., Darwish T.K., Bayoumi M.A.; et al.: Performance Analysis of Low power 1-Bit CMOS Full Adder Cells, *IEEE Transactions on VLSI Systems*, vol. 10, no. 1, February 2002.

[52] Paik, W.H., Ki H.J., Kim, S.W.: et al, Low Power Logic Design Using Push-pull Pass-transistor Logics. Int. J. electronics, 1998, vol. 84, no.5, 467-478.

[53] Wu A., Ng C.K., et al.: High Performance Low Power Low Voltage Adder, Electronic Letter 33(8)(1997) 681-682.

[54] Song, M. and Asada, K., Design of Low Power Digital VLSI Circuits Based on a Novel Pass-Transistor Logic, *IEICE Transactions on Electronics*, vol. E81-C, no. 11 p. 1740-9, November 1998.

[55] Kanus U., Meißner M., Straßer W., Pfister H., Kaufman A., Amerson R., Carter R.J., Culbertson B., Kuekes P., and Snider G., Implementations of Cube-4 on the Teramac Custom Computing Machine, *in Computer and Graphics*, Vol. 21, No. 2, pp. 199-208, 1997.

[56] Kanus U., Wetekam G., Hirche J. and Meißner M., An FPGA-based Interactive Volume Rendering System, *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, Montpellier (La Grande-Motte), France, 2002

[57] Balázs C., Fast Volume Rotation using Binary Shear-Warp Factorization, Proceedings Joint EG - IEEE TCVG Systems Symposium, pp, 145-154, Vienna, Austria, May 26-28, 1999.

[58] Meissner M., Kanus U., and Strasser W., VIZARD II: A PCI Card for Real-Time Volume Rendering, *Proc. Siggraph/Eurographics Workshop on Graphics Hardware'98*, pp. 61--67, 1998.

[59] Meissner M., Huang J., Bartz D., Mueller K., and Crawfis R., A Practical Comparison of Popular Volume Rendering Algorithms, *Proc. 2000 Symp. on Volume Rendering*, pp. 81-90, October 2000.

[60] Knittel G., The ULTRA VIS System, *Proc. Volume Visualization and Graphics Symposium*, pp. 71-80, October 2000.

[61] Sweeney J. and Mueller K., Shear-Warp Deluxe: The Shear-Warp Algorithm Revisited, *Joint Eurographics - IEEE TCVG Symposium on Visualization 2002*, Barcelona, Spain, May 2002, pp. 95-104.

[62] Wittenbrink C. M., Malzbender T. and Goss M. E.: Opacity-Weighted Color Interpolation for Volume Sampling, *Proceeding of the 1998 IEEE Symposium on Volume Visualization*, VVS 1998, October 19-20, 1998, Research Triangle Park, NC, USA.: 135-142.

[63] Magallon, M., Hopf, M., and Ertl, T. 2001. Parallel Volume Rendering Using PC Graphics Hardware. *Pacific Graphics*.

[64] Meibner, M., Grimm, S., Straber, W., Packer, J., and Latimer, D. 2001. Parallel Volume Rendering on a Single-Chip SIMD Architecture. *Proceedings of IEEE Symposium in Parallel and Large Data Visualization and Graphics.*

[65] Meißner, M. and Bartz, D.: Translucent and Opaque Direct Volume Rendering for Virtual Endoscopy Applications, In *Volume Graphics (VG 2001), Stony Brook*, 2001.

[66] Bartz, D. and Meißner, M.: Voxels versus Polygons: A Comparative Approach for Volume Graphics., In *Volume Graphics VG'99, Swansea*, 1999.

[67] Park, S., Park, S., and Bajaj, C. 2001. Hardware Accelerated Multipipe Parallel Rendering of Large Data Stream. *CS and TICAM Technical Report.*

[68] Porter, T. and Duff, T., Compositing Digital Images, *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, p.253-259, January 1984.

[69] Heirich, A. , Moll, L., Scalable Distributed Visualization Using Off-the-shelf Components, *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, p.55-59, October 25-26, 1999, San Francisco, California, United States.

[70] Kreeger, K.A., A. Kaufman, A., PAVLOV: A Programmable Architecture for Volume Processing. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Lisbon, Portugal. August 31 - September 4, 1998.

[71] Kaufman, A., Dachille, F., Chen, B. Bitter I., Kreeger K., Zhang N., Tang Q. and Hua H., Real Time Volume Rendering. *Special Issue on 3D Imaging of the International Journal of Imaging Systems and Technology*, 2000.

[72] Lin, R., A Family of High Performance Multipliers and Matrix Multiplers (US Patent pending, Dec. No. R-1265-125), 2000.

[73] Blinn, J.F., Light Reflection Functions for Simulation of Clouds and Dusty Surfaces, *Computer Graphics*, 16(3): 21-29, July 1992.

[74] Bentum M.J., Lichtenbelt, B.B.A. and Malzbender T., Analysis of Gradient Estimators in Volume Rendering, *IEEE Transactions on Visualization and Computer Grphics*, 2(3): 242-252, September 1996.

[75] Hanrahan, P., Three-Pass Affine Transformfor Volume Rendering. Computer Graphics, 24(5): 71-78, November 1990.

[76] Hoehne, K.H. and Bernstein R., Shading 3D Images from CT Using Gray level Gradients. IEEE Transactions on Medical Imaging, 5(1): 45-47, Match 1986.

[77] Gonzalez, R. and Woods, R., Digital Image Processing. Addison-Wesley, 1998.

[78] Konstantinides K., and Rasure, J., The Khoros Software Development Environment for Image and Signal Processing, *IEEE Journal of Image Processing*, to appear 1993.

[79] Ligier Y., Ratib O., Logean M. and Girard C.: OSIRIS: A Medical Image Manipulation System, *M.D. Computing Journal*, July-August 94, Vol. 11, No 4, pp 212-218.

[80] ParVox. Retrieved May 15 2002 from http://www.parvox.net/

[81] AVS. Retrieved May 15 from 2002 http://www.avs.com/software/soft_t/avs5.html

[82] Renderman. Retrieved May 15 from 2002 https://renderman.pixar.com/

[83] VolVis. Retrieved May 15 from 2002 http://www.neuro.ki.se/neuro/volvis_man/volvis_man.html

[84] VolRend. Retrieved May 15 from 2002 http://www.hlrs.de/organization/vis/people/niemeier/volrend/volrend.html

[85] Volsh. Retrieved May 15 from 2002 http://www.vets.ucar.edu/software/volsh/

[86] UltraVis. Retrieved May 15 from 2002 http://www.hpl.hp.com/ultravis/

[87] 3-D Doctor. Retrieved May 15 from 2002 http://www.ablesw.com/3d-doctor/

[88] Krot. Retrieved May 15 2002 from http://www.atnf.csiro.au/computing/software/visualisation/node7.html

[89] MRIPE. Retrieved May 15 from 2002 http://mpire.sdsc.edu/

[90] VoxBlast. Retrieved May 15 from 2002 http://www.vaytek.com/VoxBlast.html

[91] The OpenGVis project. Retrieved May 15 from 2002 http://openqvis.sourceforge.net/

[92] BOB (GVLware). Retrieved May 15 from 2002 http://biocomp.stanford.edu/3dreconstruction/software/bob.html

[93] 3Dviewnix. Retrieved May 15 from 2002 http://www.mipg.upenn.edu/~Vnews/

[94] VFleet Distributed Volume Renderer. Retrieved May 15 from http://www.psc.edu/Packages/VFleet_Home/

[95] Dr. Razz. Retrieved May 15 from 2002 http://biocomp.stanford.edu/3dreconstruction/software/dr_razz.html

[96] VrVis. Retrieved May 15 from 2002 http://www.vrvis.at/

[97] Kroros. Retrieved May 15 from 2002 http://www-vis.lbl.gov/software/khoros.html

[98] Osiris. Retrieved May 15 from 2002 http://www.expasy.org/www/UIN/html1/projects/osiris/osiris.html

[99] Simian. Retrieved May 15 from 2002 http://www.cs.utah.edu/~jmk/simian/

Appendix

Generic RAM Model

```
-- Function: generic SRAM model with separate I/O. OE and CS are active HIGH
--  WE latches data on rising edge
library ieee;
library vfp;
    use ieee.std_logic_1164.all;
    use vfp.generic_functions.all;
entity sram_core_GENERIC is
generic (
  -- entity parameterization
  address_width : integer := 0; --
  data_width    : integer := 0; --
  architecture_DNH_filename : string := "????????.dnh" -- );
port (
  address  : in  std_ulogic_vector(address_width-1 downto 0);
  CS       : in  std_ulogic;
  WE       : in  std_ulogic;
  OE       : in  std_ulogic;
  data_in  : in  std_logic_vector(data_width-1 downto 0);
  data_out : out std_logic_vector(data_width-1 downto 0)
);
begin
  -- data bus width must be integer power of 2 bits, < 64
  assert is_factor_of_32(data_width)
    report "Width of data bus must be 1, 2, 4, 8, 16 or 32 bits."
    severity warning;
end sram_core_GENERIC;
-- Architectures:
library vfp;
library ieee;
architecture original of sram_core_GENERIC is
  use vfp.host_environment_parameters.all;
  use vfp.standard_types.all;
  use vfp.generic_functions.all;
  use vfp.generic_conversions.all;
  use IEEE.std_logic_1164.all;
  constant memory_depth : integer := 2 ** address_width;
  constant num_bits_in_integer : integer :=
vfp.host_environment_parameters.num_bits_in_integer;
    attribute dnh_file_name : string(1 to 12);
  attribute dnh_file_name of sram_core_GENERIC : entity is
architecture_DNH_filename;
begin
  -- address, 0 to 31
  -- memory_depth, 0 to 7
  -- num_slices_in_integer, 4
  -- WE is inactive after the posedge until the -ve edge
  -- CS and OE are both active high
   ram_access: process (address, cs, we, oe)
    constant num_slices_in_integer : integer := num_bits_in_integer / data_in'length;
    constant memory_depth : integer := (2**(address'length)) / num_slices_in_integer;
    type integer_memory is array (0 to memory_depth-1) of integer;
    variable memory_array : integer_memory;
    variable memory_index : integer;
    variable memory_data : integer;
    variable memory_word : std_ulogic_vector(num_bits_in_integer-1 downto 0);
    variable intra_memory_index : integer;
    variable upper_bound : integer;
    variable invalid_control_signals : boolean;
    variable address_int : integer;
```

```
begin
 data_out <= (others => 'X'); -- initially set output buffers driving unknowns
 if not (cs = '1' or cs = '0') then
  invalid_control_signals := TRUE;
 elsif not (oe = '1' or oe = '0') then
  invalid_control_signals := TRUE;
 elsif not (we = '1' or we = '0') then
  invalid_control_signals := TRUE;
 else
  invalid_control_signals := FALSE;
  data_out <= (others => 'Z');
  -- set output buffers high-impedance if memory control signals are valid levels
 end if;
 if not invalid_control_signals then
  -- cs, oe and we must each be 1 or 0 for this branch to be taken
  address_int := to_integer(address);
  if cs = '1' then -- memory access possible
   if oe = '1' then  -- read or clash possible
    if we = '1' then
     -- read cycle
     memory_index := address_int / num_slices_in_integer; -- 4 := 17/4
     -- converts address into index into memory
     memory_data := memory_array(memory_index);
     -- extracts integer where addressed word is placed
     memory_word := to_std_ulogic_vector (memory_data, num_bits_in_integer);
     -- converts integer to 32-bit vector
     intra_memory_index := address_int mod num_slices_in_integer; -- 1 := 17/4
     -- address now used to get word-index of addressed word in 32-bit vector
     upper_bound := (intra_memory_index + 1) * data_out'length;
     -- defines upper index of addressed word in 32-bit vector
```

```
          data_out <= To_StdLogicVector (memory_word(upper_bound-1 downto
upper_bound-data_out'length));
          -- To_StdLogicVector is IEEE std_logic_1164, drives data bus with addressed
memory word
        elsif we = '0' then
          assert false
            report "oe and we are both active."
            severity warning;
        end if;
      elsif oe = '0' then -- oe is inactive, only write possible
        if (we = '1') and we'event then
          -- write cycle
          memory_index := address_int / num_slices_in_integer;
          -- converts address into index into memory
          memory_data := memory_array(memory_index);
          -- extracts integer where addressed word is placed
          memory_word := to_std_ulogic_vector (memory_data, num_bits_in_integer);
          -- converts integer to 32-bit vector
          intra_memory_index := address_int mod num_slices_in_integer;
          -- address now used to get word-index of addressed word in 32-bit vector
          upper_bound := (intra_memory_index + 1) * data_in'length;
          -- defines upper index of addressed word in 32-bit vector
          memory_word(upper_bound-1 downto upper_bound-data_in'length) :=
to_StdULogicVector (data_in);
          -- inserts data into extracted memory word
          -- ... now convert whole word to integer & place back in integer array
          memory_array(memory_index) :=
to_integer(to_twos_complement(memory_word));
        elsif we = '1' then
          -- both we and oe not active, nowt happens
        elsif we = '0' then
      -- we active, nowt happens until posedge of WE
     end if;
    end if;
   else
    -- memory not selected
   end if;
  else
   assert false
    report "Control signals are not active, memory is not selected."
    severity note;
  end if;
 end process;
end original;
```

Memory controller model (RAM chip file) for RD and WR.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity RamChip is
 port (Address: in Std_logic_vector(3 downto 0);
       Data: inout Std_logic_vector(7 downto 0);
       CS, WE, OE: in Std_logic);
end;
architecture Behaviour of RamChip is
begin
 process(Address, CS, WE, OE)
   subtype Byte is Std_logic_vector(7 downto 0);
   type Mem is array (0 to 15) of Byte;
   variable Memory: Mem := (others => Byte'(others=>'U'));
 begin
   Data <= (others => 'Z');
   if CS  = '0' then
     if OE = '0' then                    -- Read operation
       Data <= Memory(To_Integer(Address));
     elsif WE = '0' then                 -- Write operation
       Memory(To_Integer(Address)) := Data;
     end if;
   end if;
 end process;
end;
```

Glossary

2C-adder-two's complement adder

2D-Two-Dimensional

3D-Three-Dimensional

ASIC-Application Specific Integrated Circuits

BE-Booth Encoder

BIST-Built In Self Test

CAT-Computed Axial Tomography

Cin-Carry in

CMC-Canadian Microelectronics Corporation

CMOS-Complementary Metal Oxide Semiconductor

Cout–Carry out

CPL-Complementary Pass Transistor Logic

CPU-Central Processing Unit

CS-Carry-Select

CT-Computer Tomography

CUT-Circuit Under Test

DEMUX-DE-MUltipleXer

DFT-Design For Testability

dpr-dual port

DSP-Digital signal processing

ECC RAM- Error Code Correction RAM

ECI-External Communication Interface

EM-Enhanced Memory

FF-Flip Flop

FIFO-First-In-First-Out

FMT-Fully Merged Technique

FPGA-Field Programmable Gate Array

fps-frames per second

FS CPL-Full Swing Complementary Pass transistor Logic

FS-Full Swing

HVS-Human Visual System

I/O pin-input/output pin

IPP-Inner Product Processor

LCM-Laser Confocal Microscopy

LSB-Least Significant Bit

MA-Merged Arithmetic

MCM-Multi-Chip Module

MMP-Matrix Multiplication Processor

MOSFET-Metal Oxide Semiconductor Field Effect Transistor

MRI-Magnetic Resonance Imaging

MSB-Most Significant Bit

MUX-MUltipleXer

NFS CPL-Non Full Swing Complementary Pass transistor Logic

NFS-Non Full Swing

NMOS-N-channel Metal Oxide Semiconductor

NR–Normal Register

ORA-Output Response Analyzer

PAVLOV-Parallel Array for VoLume prOcessing and Viewing

PGA-Pin-Grid-Array

PMOS-M-channel Metal Oxide Semiconductor

PMT-Partially Merged Technique

PPG-Partial Product Generator

PP-Partial Product

PTFF-Phase Clocking FF

R/O-Read Only

R/W-Read/Write

RD-ReaD

REG REGister

RTL-Register Transfer Level

SCLK-System CLocK

S-Sum

SWF-Shear-Warp Factorization algorithm.

TG-Transmission Gate

TPG-Test Pattern Generator

TSMC-Taiwan Semiconductor Manufacturing Company

TSPC-True Single Phase Clocks

UP-MicroProcessor

VIS-Volume Imaging System

VRS-Volume Rendering System

W/L ratio-Width/Length ratio

WR-WRite